April 1987

Report No. STAN-CS-88-1204

Thesis

AD-A198 710

DTIC FILE COPY

2

Exploiting Constraints in Design Synthesis

by

Joseph Jeffrey Finger

Department of Computer Science

Stanford University

Stanford, California 94305

DTIC
S ELECTE D
AUG 0 4 1988
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION: unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release: Distribution Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S): STAN-CS-88-1204

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION: Computer Science Department

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS (City, State, and ZIP Code): Stanford University, Stanford, CA 94305

7b. ADDRESS (City, State, and ZIP Code)

8a. NAME OF FUNDING / SPONSORING ORGANIZATION: DARPA

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd. Arlington, VA 22209

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Exploiting Constraints in Design Synthesis

12. PERSONAL AUTHOR(S): Joseph Jeffrey Finger

13a. TYPE OF REPORT

13b. TIME COVERED FROM ________ TO ________

14. DATE OF REPORT (Year, Month, Day): April 1987

15. PAGE COUNT: 128

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The class of design synthesis problems encompasses a wide spectrum of common encountered problems, including robot planning problems, synthesis of electronic circuits, chemical synthesis, genetics experiment design, and computer program synthesis. This thesis is in two main parts, both dealing with design synthesis. The first part is the RESIDUE METHOD, an abductive approach to design synthesis, and the second is SUPERSUMPTION, a generalization of consistency checking of partially completed designs.

The RESIDUE METHOD synthesizes designs by reduction of the design goal to another, primitively achievable goal. The reduced goal must be consistent with known facts about the world, must be sufficient to achieve the original goal, and must be a conjunction of formulas from a language of primitively achievable formulas. The RESIDUE METHOD expresses the design goal, the final design, and all intermediate designs as formulas of first-order logic. Soundness and completeness results are given for two resolution-based residue procedures.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

EXPLOITING CONSTRAINTS IN DESIGN SYNTHESIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DTIC
COPY
INSPECTED
6

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

By
Joseph Jeffrey Finger
March 1987

© Copyright 1987

by

Joseph Jeffrey Finger

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Genesereth

Michael R. Genesereth
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bruce G. Buchanan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Richard J. Waldinger

Richard Waldinger
(SRI International)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

The class of design synthesis problems encompasses a wide spectrum of common encountered problems, including robot planning problems, synthesis of electronic circuits, chemical synthesis, genetics experiment design, and computer program synthesis. This thesis is in two main parts, both dealing with design synthesis. The first part is the ***Residue Method***, an abductive approach to design synthesis, and the second is ***supersumption***, a generalization of consistency checking of partially completed designs.

The Residue Method synthesizes designs by reduction of the design goal to another, primitively achievable goal. The reduced goal must be consistent with known facts about the world, must be sufficient to achieve the original goal, and must be a conjunction of formulas from a language of primitively achievable formulas. The Residue Method expresses the design goal, the final design, and all intermediate designs as formulas of first-order logic. The usual approach in deductive synthesis has been to express designs as a single term of composed state transformation functions. Expression of designs as a formula rather than a term simplifies synthesis of non-linear plans, allows postponing imposition of ordering constraints, and allows one to reason directly about the proposed design. Soundness and completeness results are given for two resolution-based residue procedures.

Supersumption is an attempt to exploit the consistency requirement in order to accelerate synthesize of designs. Not only is consistency of a partially completed design checked, but additional "ramifications" may be derived that must be true for the partial design to remain consistent. By making sure that the ramifications are not violated, one avoids searching parts of the search space that do not contain legal designs. In addition, knowing ramifications may make additional search control heuristics directly applicable. The process of imposing additional constraints on a subgoal is called "supersumption." Two phenomena are described by which supersumption can speed up the search — use of ramifications as better generators of candidates than the unadorned goal, and use of ramifications as filters to quickly eliminate inconsistent designs. Two resolution-based methods for deriving ramifications are given, along with soundness and completeness results.

Acknowledgements

First and foremost, I would like to express my gratitude and indebtedness to my thesis advisor Michael Genesereth. Besides being my teacher, Mike has been a source of optimism, of new ideas and of sharp criticisms. He has encouraged me at every step of the way and stuck it out with me to the end.

Bruce Buchanan and Richard Waldinger, my other Reading Committee members, both went to great lengths to help me, and they bent over backwards to get my thesis read in time. I am very grateful to both, and wish I would have made more and better use of all of their talents and able suggestions.

Many thanks to Gio Wiederhold for serving on my Orals Committee.

Zohar Manna provided me with more than a little sound and very necessary advice, and was probably more effective than anyone at giving me a gentle (or not so gentle) shove when it was needed.

The rest of the Logic Group has provided a receptive and patient sounding board over the years. Thanks in particular to Tom Dietterich, Matt Ginsberg, Russ Greiner, Jock Mackinlay, Jeff Rosenschein, Narinder Singh, Dave Smith, Devika Subramanian, and Richard Treitel.

The long, hard process of completing a disseration would have been completely impossible without the support of many friends over the years. I would like to especially mention each of the following: Avron Barr, Jim Bennett, Ora Biran, Stephanie Buchholz, Batia Eshel, Donna Goldberg, Jacques Goldberg, Suzanne Jacobs, Don Katcoff, Ira Machefsky, Pam Machefsky, Alissa Nordlicht, Dan Perkins, Anat Rafaeli, Sheizaf Rafaeli, Beth Rosenschein, Jeff Rosenschein, Shmuel Shaffer, Nahum Silverman, Dick Sites, Debbie Wenocur, Michael Wenocur, and Eli Yaacobi.

My years at Stanford would have been very different and much poorer were it not for my office mates — fellow occupants of the Bozo Bus – in Cedar Hall and later in Margaret Jacks Hall: Paul Cohen, John Kunz, Jock Mackinlay, Jeff Rosenschein, and David Smith.

Finally, I want to thank my family for all their support during these years. Thanks

go to my parents Julia and Joseph Finger, my sister and brother-in-law Tassie and Steve Bielsky, my aunt and uncle Rosalie and Alfons Salinger, and to my grandmother Gertrude Levy Finger, who had hoped very much to see me complete this project.

Contents

List of Figures

Chapter 1

Introduction

> *Everyone designs who devises courses of action aimed at changing existing situations into preferred ones. The intellectual activity that produces material artifacts is no different fundamentally from the one that prescribes remedies for a sick patient or the one that devises a new sales plan for a company or a social welfare policy for a state.*
>
> *Herbert Simon,* The Science of Design [1]

1.1 Overview

Robot planning, genetic synthesis, chemical synthesis, circuit design, and program synthesis are but a few examples of *synthesis* or *design* problems. In each of these domains the design process is that of *finding a composition of known types of components to form a whole meeting given specifications.* Almost from the beginning of the study of Artificial Intelligence in the 1950's, researchers have sought to automate the process of design synthesis. A great many systems have been developed, some general and others special-purpose, some formal in approach and others less formal.

This dissertation attempts to find a formal framework that captures the notion of the design process as *making and using a possibly conflicting series of design decisions to restrict the set of candidate designs.* Three main facets of the design process are explored:

1. **Residue**, a deductive framework for synthesis in which designs are represented as sets of formulas.

[1] From *The Sciences of the Artificial*, Second Edition, The MIT Press, Cambridge, Massachusetts, 1969,1981, page 129.

2. **Supersumption**, a technique for reformulation of design goals via added constraints.
3. **Finding Ramifications**, procedures for generating additional constraints that must be satisfied in order for a design to be found. These constraints are found as part of checking the consistency of a partially completed design.

1.1.1 Residue — Representation of Designs as Formulas

To solve a design synthesis problem is to map one specification into another. One starts with a specification G (the *goal*) of what needs to be true of the object designed. A new specification **D** (the *design*) is sought, such that **D** not only guarantees that G will be met, but such that it is possible to implement **D**, and where **D** is specified at such a level that the intended implementor needs no further instruction.

Chapter 2 presents such a formulation of design synthesis expressed in first-order predicate calculus. It is assumed that there is a consistent set **W** of axioms describing the world and that the goal G is expressed as a single formula. Furthermore it is assumed that there is a language $\mathcal{A}$ of "assumable" formulas, that is, all formulas that specify an instruction simple enough for the implementor to carry out without further instruction. It is assumed that there is an algorithm that decides in negligible time whether an arbitrary formula is in $\mathcal{A}$ or not. For convenience, let us also assume that a conjunction of assumable formulas is assumable, that is, if $A_1, \ldots, A_n \in \mathcal{A}$ then $A_1 \wedge \ldots \wedge A_n \in \mathcal{A}$.

To be a legal design specification a set of formulas $\mathbf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_m\}$ must be such that

1. $\mathbf{W}, \mathbf{D} \models \mathsf{G}$ (The design achieves the goal),
2. $\mathbf{W} \cup \mathbf{D}$ is satisfiable (The design is consistent with the world model), and
3. For all $\mathsf{D}_i \in \mathbf{D}$, $\mathsf{D}_i \in \mathcal{A}$. (The design is expressed in terms of the design primitives).

Any set **D** with the above properties is called a *residue*.[2] Sometimes it will be convenient to view **D** as a single formula $\mathsf{D} = \mathsf{D}_1 \wedge \ldots \wedge \mathsf{D}_m$. Each fact D_i can be seen as a *commitment* or *constraint* upon the design, that is, a *design decision* on the part of the designer.[3]

[2]One might call this approach to design an *abductive* approach, though not without some confusion. The term *abduction* or *apogage* dates back to Aristotle. An abduction is "a syllogism whose major premise is known to be true but whose minor premise is merely probable." (*The Encyclopedia of Philosophy*, Macmillan Publishing Co., Inc. & The Free Press, New York, 1967, page 5-57). See also Hempel [42]. Charniak and McDermott [16] use the term in a similar way in describing generation of explanations. In addition, Charles Sanders Peirce (1839-1914) [36] used "abduction" to mean the "creative formulation of statistical hypotheses" (*Encyclopedia of Philosophy*, page 4-176).

[3]The idea of a design being built up as a sequence of decisions or constraints is not a new one, in fact, virtually any search process can be viewed as a sequence of implicit or explicit decisions. Heuristic Dendral [8], REF-ARF [24] and Stefik's MOLGEN planner [93,92] are important examples of synthesis procedures in which the decisions are explicit.

Representation of a design as a set of assumable facts is called the *Residue Approach* in this work.

In logic, a term denotes an individual of the *domain* (or *universe of discourse*, as it is sometimes called); a formula denotes a proposition about the world. In the residue approach, the domain consists of possible design components and parameter values. Terms denote such components or values, and formulas express propositions about those terms. For example, to design a combinational circuit, the domain might be the possible NAND-gates, wires, inputs of NAND-gates, and outputs of NAND-gates. Synthesis of RC-circuits via the residue approach might entail a domain of wires, resistors, capacitors, resistors, capacitances, and resistances. Decisions about the design would be expressed as formulas denoting propositions about the wires, resistors, resistances, etc.

Example 1.1 Consider a planning problem in which the designer has decided that (1) a $\mathsf{puton}(A, B)$ action will be executed and (2) a $\mathsf{puton}(B, C)$ will be executed. These two decisions can be represented via the two formulas:

$$\mathsf{Execution}(\mathsf{puton}(A, B), T_1)$$
$$\mathsf{Execution}(\mathsf{puton}(B, C), T_2).$$

Puton is a function mapping two blocks into an action. Execution is a relation on an action a and a time t and denotes the proposition that action a will be executed at time t. Note that the above two facts in no way determine whether time T_1 is before or after time T_2.

The Single-Term Approach Deductive synthesis research (See, for example, Green [32], Waldinger and Lee [99], Luckham and Nilsson [54], Manna and Waldinger [58,59], and Wos, et al [106].) has traditionally represented designs as a single term, that is, both completed designs and incomplete designs are represented as a composition of functions. For example, a robot plan to put block B on block C and then put block A on block B might be represented as

$$\mathsf{puton}(A, B, \mathsf{puton}(B, C, S_0)), \qquad (1)$$

where S_0 is the initial state of the system, and puton is a function mapping two blocks and a state to a state. The above approach of representing the design via a single term will be called the *single-term approach*.

The single-term approach requires that all designs, both completed and incomplete, be a part of the domain. It is not difficult to imagine finding a set of functions with which to

build terms expressing completed designs, but a set of functions for expressing all desired incomplete designs is more problematic.

Example 1.2 Consider the example from (1) of the robot plan expressed as $\mathsf{puton}(A, B, \mathsf{puton}(B, C, S_0))$. If, during the design process, it is determined that the plan should contain a $\mathsf{puton}(A, B)$ action and also a $\mathsf{puton}(B, C)$ action without specifying their order, it is not clear how to express the incomplete plan as a single term. Instead, one must choose either the term $\mathsf{puton}(A, B, \mathsf{puton}(B, C, S_0))$ or the term $\mathsf{puton}(B, C, \mathsf{puton}(A, B, S_0))$.

The difficulty in the above example is that the set of *completed* designs is easily represented as a single term, but the set of desired partial designs is much bigger and not as ammenable to representation as a single term. Note that in order to express the desired information, the partial design had to be constrained more than necessary.

Advantages of the Residue Approach The thesis claims that it is beneficial to represent designs, especially incomplete designs, as sets of formulas rather than single terms. The following reasons will be elaborated in Chapter 2:

1. As illustrated in Examples 1.1 and 1.2, the residue approach is more expressive than the single-term approach; many design decisions can be expressed easily in the residue approach, but can only be expressed via a stronger constraint in the single-term approach. In such a case, the lack of expressiveness of the single-term approach can result in unnecessary backtracking.[4] Attempts to avoid such difficulties in the single-term approach are frought with difficulties.

   It should be noted that syntactically, any set of formulas can be encoded as a composition of functions — one need only define a new $(n+1)$-ary function for each n-ary relation and connective in the language. Thus, to say that single terms are inherently less expressive than sets of formulas is in some sense incorrect. On the other hand, such an encoding of a set of formulas begs the question. Instead of denoting a set of state transformations constituting a prescription for a design, such a composition of functions would denote any design for which all the encoded propositions hold — a perverse way of taking the residue approach.

[4] Similar ideas were expressed in Stallman and Sussman's EL [90] and in the *least commitment cycle* of Stefik's planning engine [93,92].

2. Just as incomplete designs are better represented as sets of facts than as single terms, one might desire a greater expressness for complete designs as well. In Example 1.2, for example, it might not matter which action comes first, or even whether they are executed in parallel. As such, it is usually desirable not to commit to one ordering or the other (as demanded by the single-term approach).

3. For reasoning about control strategies, performing consistency checking or finding supersumptions (presented in Chapters 3 and 4), it is important to be able to reason *about* the design. An inference engine can reason directly from a set of facts using well-understood mechanisms of logic. In contrast, reasoning about a single term is an ad hoc process.

4. There are problems for which the full expressiveness is not needed, that is, the expressiveness of single-term approach is perfectly adequate. For such cases, the residue approach can be used on these problems as efficiently as the single-term approach.

The above formulation of design problems is not operational — there must be a procedure for finding residues. Chapter 2 presents two such procedures for generating residues, Resolution Residue and Ordered Residue. In addition, it proves appropriate completeness results for the two procedures.

1.1.2 Supersumption — Faster Solution via Added Constraints

In solving design problems via various residue procedures, it was noticed that the systems were not able to take advantage of information gained during consistency checking. Rather than just knowing that a design is consistent, it was desired to know what conditions will have to hold if the design is to *remain* consistent. Such a condition can then be incorporated into the goal to avoid needless search.

Supersumption is a such technique for incorporating these conditions, reformulating goals in such a way that the cost of solving the reformulated goal can be less than the cost of solving the original goal.

Example 1.3 Suppose a personal computer owner has numerous 256 Kbyte floppy disks, a 50 Mbyte hard disk, and the goal,

"Find all disk files larger than 1 Mbyte."

Due to its size, no such file could possibly exist on a floppy disk, but would have to be on the hard disk. The reformulated goal,

"Find all *hard* disk files larger than 1 Mbyte"

would (1) have the same set of solutions as the original goal and (2) presumably be cheaper to solve than the original goal in that it avoids enumerating the floppy disk files.

In the above example, the reformulated goal has the same set of solutions as the orignal, but this need not be the case:

Example 1.4 Suppose the computer owner of Example 1.3 is given the goal,

"Find a disk file larger than 200 Kbytes."

If the owner knows that

"Most files larger than 128 Kbytes are on hard disk,"

he might reformulate the goal as,

"Find a *hard* disk file larger than 200 Kbytes."

The reformulated goal may eliminate some solutions to the original goal, but (1) some solution is likely to be found, and (2) the reformulated goal is presumably cheaper to solve than the original goal in that it avoids enumerating the floppy disk files. The owner may choose to remember the original goal in case no solution is found for the reformulated goal.

In both of the above examples, an original goal was constrained by additional requirements. In Example 1.3 the additional requirement (that the file be on the hard disk) is logically implied by the goal G, any design decisions **D** (none in this problem), and the known facts **W** about the world. The subclass of logically implied additional constraints will be called *ramifications*. In contrast, the additional requirement in Example 1.4 is not a logical implication of the goal and the known facts about the world; it is likely to be true of any solution, but some solutions of the original goal may not be solutions of the reformulated goal. Such a constraint will be called a *probable constraint*.

The above examples can be characterized by taking a goal G and reformulating the goal to be $\mathsf{G} \wedge \mathsf{A}$. Chapter 3 develops the above notion of *supersumption*, the conjoining

of additional constraints to an existing goal or subgoal. Supersumption may be done with any additional constraint, either necessary (that is, a ramification) or not, but the thesis concentrates on ramifications alone.

The above reformulations via supersumption are not sufficient to bring about a speedup, however. In additional there must be a *strategy* for taking advantage of the added constraint.

Example 1.5 Suppose the goal specification of Example 1.3 is written as

$$\text{Find an } x \text{ such that: } \mathsf{File}(x) \wedge (\mathsf{Size}(x) > 1\text{Mbyte}).$$

Addition of the ramification $\mathsf{Hard\text{-}Disk\text{-}File}(x)$ produces the goal

$$\text{Find an } x \text{ such that: } \mathsf{File}(x) \wedge (\mathsf{Size}(x) > 1\text{Mbyte}) \wedge \mathsf{Hard\text{-}Disk\text{-}File}(x),$$

but says nothing about how to use $\mathsf{Hard\text{-}Disk\text{-}File}(x)$ in speeding up the search. A speedup is obtained only by specifying a processing method such that the x such that $\mathsf{Hard\text{-}Disk\text{-}File}(x)$ are enumerated rather than the x such that $\mathsf{File}(x)$.

In this thesis, the only strategy considered will be reordering the conjuncts of a conjunctive goal, the subject of research by David E. Smith [86,85].

1.1.3 Ramifications — Generalizing Constraint Propagation

As mentioned in Section 1.1.2, an important class of supersumptions is supersumption with *ramifications*, that is, with constraints logically implied by the current goal G (as opposed to the original goal), the world model **W**, and the design decisions **D** to date. A ramification N of a goal G and design decisions **D** is a formula such that the goal has no solution (given the design decisions to date) for which the ramification does not hold as well. In other words, if $\mathbf{D} = \mathsf{D}_1 \wedge \ldots \wedge \mathsf{D}_m$, then N is a formula such that $\mathbf{W} \models (\mathbf{D} \wedge \mathsf{G}) \supset \mathsf{N}$.

Example 1.6 In Example 1.3, a personal computer owner has numerous 256 Kbyte floppy disks, a 50 Mbyte hard disk, and the goal,

"Find all disk files larger than 1 Mbyte."

As before, no such file could possibly exist on a floppy disk, that is, the condition

"the files are all on hard disk"

is a ramification of the above goal; there is no design for which the ramification does not hold.

Ramifications and Consistency Checking Ramifications can be generated as a part of checking consistency of a design, complete or incomplete. If a partial design is inconsistent, there is no consistent complete design incorporating all the decisions of the partial design, that is, some design decision has to be relaxed. If consistency checking were free, it would always pay to know whether the current partial design is consistent. As it turns out, complete consistency checking is in general intractable and can only be approximated. If a problem is expressed in first-order logic, then inconsistent designs can be proven to be inconsistent, but not in a bounded amount of time. In general, there is no way to be sure that a design is consistent.

Inconsistency can be proven by showing that **false** logically follows from (1) **D**, the design decisions to date, (2) G, the goal at hand, and (3) the world model **W**. In other words, if **false** is a ramification of the current G and **D**, then the current partial design is inconsistent. In the process trying to deriving **false**, one derives other ramifications. In Chapter 4 it is shown exactly what other ramifications will be derived by various procedures for checking consistency.

By recording ramifications, one gets additional information in answering the question, "Is the design to date consistent?" Instead of a yes/no answer, one receives an answer of "No," or an answer, "Yes, the partial design is consistent, but ramification A must hold for all complete designs incorporating this partial design."

Ramifications and Constraint Propagation Finding ramifications may rightfully be viewed as a generalized form of constraint propagation, a way to fit constraint propagation into a deductive approach. Constraint propagation is usually thought of as a specific inferences to be carried out when certain triggering conditions are met, for example in Waltz's line labelling program [103,102], in Stallman and Sussman's EL [90], or in Stefik's constraint posting [93,92]. In this research ramifications are found via the general mechanism of forward inference. Instead of making a specific inference, one faces a search problem with difficult questions of (1) how to search and (2) what constitutes a useful ramification.

1.2 Approach and Scope of the Thesis

1.2.1 Deductive Synthesis

The approach taken in this thesis is that of *deductive synthesis*. *Deduction* is defined as, "the act or process of reasoning, especially a logical method in which a conclusion necessarily

follows from the propositions stated."[5] *Deductive synthesis* refers to the construction of an implementable specification of an object as part of the proof of a theorem that the object meets a certain (possibly abstract) specification. In this thesis, the world model **W**, the design goal G, and the design **D** itself are all represented as formulas or sets of formulas in first-order predicate calculus, but they need not be; there is no reason not to do deductive synthesis in some other logic (See, for example Konolige [46]). There are also numerous alternatives to *deductive* synthesis systems (See, for example, Burstall [11], Manna and Waldinger [60], Barstow [2], and Green [33]). The fact that a system is not deductive does not mean that its output is more or less believable than a deductive system's output — a deductive system's deductions are only as good as its axiomatization of the world.

Taking a deductive approach implies a *declarative* approach to knowledge representation. In declarative approaches, there is a generally strict separation of control of search and inference. Instead of expressing algorithms procedurally, one expresses a particular strategy to control the search through the space of possible inferences, where the inference engine is making inferences (sound or otherwise) from some body of information about the world. The declarative approach has its roots in a number of AI systems based upon theorem proving. Some notable examples are QA3 [32], STRIPS [25], and FOL [26]. Pat Hayes' early papers [39,38,37] on declarativism were also seminal. A declarativist view is part and parcel of logic programming, and such a view has been embodied in such systems as Prolog [78] and MRS [27,79]. See also Kowalski [48,47] for early expressions of declarativism as it related to logic programming.

In design synthesis, one hopes for a number of advantages in separating control knowledge and world knowledge:

1. Knowledge about design components is expressed independent of its use. The set of facts about design components need only reflect what is true in the world, and can be checked independent of the design engine.

2. A different design engine, i.e., different control strategy or different inference engine, might use the same body of knowledge about the particular design domain. Research on control of inference becomes applicable to particular problems of a declarative system.

3. Evolution of knowledge about the domain does not require changes in the design engine (although changes for the sake of efficiency might be advisable).

4. By compiling the set of inferences made by a particular design engine on a particular set of rules, one can achieve the same speed as with procedural systems.

[5] *The American Heritage Dictionary*, Dell Publishing Co., Inc., New York, NY, 1983.

The above (or similar) claims for declarative representation have often been heard in the declarative/procedural controversy that has been raging, on and off, since the early 1970's. After fifteen years of dispute, the final results and outcome of the controversy are not entirely clear. See, for example, Winograd [104], or the *Handbook of Artificial Intelligence* [1] for discussions of the Declarative/Procedural Controversy. See McDermott [68] for the current view of a discouraged declarativist.

1.2.2 Number of Solutions to a Problem

There are many sorts of goals[6] for which a problem solver might be asked to find a solution. Examples might include:

- **Does** there exist a file written in the last hour?
- Find **all** files written in the last hour.
- Find **any** file written in the last hour.
- Find **4** files written in the last hour.
- *Find **the largest** file written in the last hour.*

In the relational database literature, the standard problem is to find *all* tuples meeting some criterion, for example, all files written in the last hour, or all flights between San Francisco and Denver leaving between 1 p.m. and 4:30 p.m. on January 10. In contrast, design problems usually require only one solution. We do not care about finding *all* circuits to shift bits in a 24 bit word, in fact, there is usually an infinite number of solutions to such problems, anyway. Instead, one must find at least *one* solution.

The residue approach is geared toward finding single solutions. By using an agenda mechanism for its search, a residue procedure can also find multiple solutions. To find the next solution, the procedure can simply be restarted with the agenda in the state where it left off. To find *all* solutions, the residue procedure must be called until its agenda is

[6]The word *goal* is commonly used in at least two ways. The goal might be a desired final state, or it might be the path by which one arrives at that final state. Sometimes the final state is described in terms of that path, as is the case in the single-term approach. In Chapters 2-5, goals are the former, that is, a description of the final destination or state in which one desires to be. In this chapter, the word *goal* is used a bit more loosely. For example, in the "goal types" enumerated below in the text, the notion of goal is of yet another variety, that is, an expression of a task to be carried out; it falls into neither the first nor the second notion of goal described above. Since the proper notion of goal is clear from the context in this and other examples of the present chapter, the different notions of goals will not be further distinguished. In future chapters, goals will be specified according to the first notion, that is, a description of the desired state for which we may find zero, one, or many solutions.

exhausted or it is known that all solutions have been found.[7] Thus, for all of the above goal types except the last, an agenda-based residue procedure can provide answers by simply being called the appropriate number of times. The last goal type above is not covered in this work because it requires an additional mechanism for looking over the set of all solutions generated.

The notation for queries used in the rest of this thesis is adapted from D. E. Smith [85] and is as follows: Queries are expressed in the form "find n v: g," where g is a formula containing zero or more free variables, n is the number of solutions desired, or "all" if all solutions are desired, v is the subset of the free variables in g for which values are required (the other free variables are assumed to be existentially quantified). If v is the entire set of free variables in g, it will be omitted. Most queries will be expressed simply as a formula g; this means that it is assumed that (1) all free variables are of interest, and (2) it is irrevelant to the discussion whether one, some, or all solutions are desired.

Whether one needs all answers or just a single answer is important in considering search methods. For problems requiring all solutions, the order in which the solution space is searched is not important. As long as there is a possibility of finding an answer in some corner of the space, that corner must be searched, and it does not matter whether it is searched first or last. On the other hand, if one only needs to find a single answer to a problem, it is best to look where an answer is most likely to be found quickest. In fact, to find any proper subset of all the solutions, it is best to look first where answers will be found the quickest. To illustrate the above phenomenon, consider the following example:

Example 1.7 Suppose

$$\mathsf{G}(x) = \mathsf{File}(x) \wedge \mathsf{Name}(x, \mathsf{chess}) \wedge \mathsf{Executable}(x),$$

that is, x is an executable file named chess. Suppose also that it is known that chess is a game, and that *most* executable files for games are on directory /usr/games. If the goal is "Find all x: $\mathsf{G}(x)$," it does not help to know that most of the answers will come from /usr/games. On the other hand, if the goal is "Find 1 x: $\mathsf{G}(x)$," or "Find 10 x: $\mathsf{G}(x)$," then it would be smart to begin the search on /usr/games.

In finding a single solution for a conjunctive goal, all the solutions for a single conjunct will be required in the worst case. Even so, it pays to find the easiest solutions first since on the average not all the solutions will have to be generated.

[7]Note also that the order of the search can affect whether the search provides all answers or whether it loops.

1.2.3 The Qualification and Ramification Problems

Two major problems in design synthesis are the well-known *qualification problem* (McCarthy [65]) and its dual problem, the *ramification problem* (discussed below). At the outset, let us note that these problems are only indirectly addressed by using a declarative approach. It has long been recognized that for any real-world design component to work as expected, there are an unbounded number of prerequisites that must be fulfilled. This problem is usually known as the *qualification problem.* The classic example is the "potato in the tailpipe." A rule might say, "If a car has gas, turning the key in the ignition will cause the car to start." The above rule names one prerequisite, namely, that the car has gasoline. If, however, a potato can be put in the tailpipe of the car, the above rule is no longer correct.[8] The rule might be fixed to include a "no potato in the tailpipe" prerequisite, but one can always find another heretofore unmentioned and obscure prerequisite that might be violated in the real world.

A declarative approach to synthesis does not directly address the qualification problem. It may explicitly name known prerequisites for a design component to behave as expected, but in no way does it tell one what additional qualifications to a rule must be made. Thus, via sound rules of inference, one can "prove" that a given design will meet its specifications, but the proof has meaning only insofar as the descriptions of design components and their behaviors actually describe the world.

A second problem endemic to design synthesis, whether declarative or otherwise, is the *ramification problem*, a dual problem to the qualification problem. In general, a given design not only depends upon an unbounded number of prerequisites (the qualification problem), but *it has an unbounded number of postrequisites*, that is, of ramifications. Consider the goal of removing a single file from a given directory. Removing all the files from that directory meets the goal specification, but is probably unacceptable.

In general, there is no way for a given goal specification to name all the postrequisites that should *not* be true of a given design, and a declarative and/or deductive approach does not directly address this problem any more than other approaches.[9]

[8] One cannot help but note that the above "potato in the tailpipe" scenario is wrong, though well-established in the AI literature. The car *will* start; it just will not run for very long (assuming yet other conditions like "no holes in the exhaust system"). See *Beverly Hills Cop* for demonstration of the more realistic scenario.

[9] The residue approach of Chapter 2 *does* provide a convenient hook for checking for some undesirable postrequisites, namely, the mechanism of consistency checking.

1.2.4 Best Fit Designs

This thesis considers the case of designs that meet a given goal specification. An interesting class of problems are those in which the goal is overspecified — even though no design is expected to meet the entire goal specification, one desires the design coming closest to doing so according to some metric. Examples of such problems are studied in Barbara Hayes-Roth and Frederick Hayes-Roth's errand planning work [41] and in PROTEAN (Buchanan et al [10,40]).

Although at first glance a deductive approach seems antithetical to "solving" overspecified problems, it need not be. Given a suitable metric for how much of a given goal has been achieved by a certain design, one could consider deductive synthesis of solutions to parts of an overspecified goal. Such problems are not considered further in this work, and to the best of the author's knowledge, deductive approaches has not been explored for overspecified goals.

1.3 Reader's Guide

Each of the main ideas presented in this chapter is covered in detail by a chapter of the thesis. Chapter 2 covers residues and residue procedures, Chapter 3 covers supersumption, and Chapter 4 is on finding ramifications. Each is designed to be as independent of the others as possible. Chapter 2 and Chapter 3 both stand as independent units, and Chapter 4 has only a slight amount of dependence on Chapter 3. Chapter 5 concludes with a summary of results, limitations and future work.

Chapter 2

Residue

This chapter presents design synthesis as a problem of finding *residues*. Section 2.1 discusses design synthesis problems and their scope, after which Section 2.2 is devoted to a definition of design synthesis as a problem in first-order logic. Section 2.3 describes procedures for generating residues, examples of which are Ordered Residue, presented in Section 2.4, and Resolution Residue, presented in Section 2.5. Ordered Residue, based on Horn Clause Resolution, is the residue procedure used in the rest of the thesis. It is presented along with a limited completeness result. Resolution Residue, based on full binary resolution, is presented with stronger completeness results. Section 2.6 deals with constructing values for existentially quantified variables in a goal specification. Section 2.7 compares and contrasts the residue approach with the "single-term approach" that has been used in most deductive synthesis systems to date. Section 2.8 compares and contrasts the residue approach to a number of other recent systems, and Section 2.9 presents conclusions of this chapter.

2.1 Design Synthesis

There exists an almost unlimited variety of *synthesis* or *design* problems. Circuit design, program design, robot planning, building design, chemical synthesis, and genetic synthesis are a tiny subset of the multitude of problems in which primitive building blocks are composed in such a way that the result meets a set of output specifications.

The terms *synthesis problem*, *design problem* and *design synthesis problem* have been used in numerous contexts. The present work makes no distiction among them (and uses them interchangably), but it assumes the following general scenario:

> There are two agents, a *designer* and an *implementor*, not necessarily distinct. The designer is given a specification (the *goal*) of the requirements of the

> object (material or otherwise) the implementor will implement.[1] In addition, the designer has information about components at the implementor's disposal, and what the implementor is and is not capable of carrying out. The objective of the designer is to map the original specification to another specification (the *design*) such that the new specification (1) describes an implementation meeting the original specification, and (2) describes the implementation process in sufficient detail that the implementor is capable of carrying it out. [2]

As emphasized in the above paragraph, **design synthesis is a mapping of one set of specifications to another**; the design itself is merely another set of specifications. A useful design will usually be a set of specifications at less abstract level than the specifications input to the design engine.

In any particular case, the criterion for what constitutes an adequate design specification is a somewhat arbitrary. Usually, a given domain will have a language and set of conventions specifying exactly what constitutes an adequate design. In building design, for example, there is a standard set of drawings that must be submitted. These drawings do not specify the construction or construction procedure to any ultimate level of detail. Instead, there is a set of conventions as to the necessary level of detail. The architect *does* have to specify the dimensions and building materials for the walls of a house. In general, he *does not* specify what size nails to use or the order in which the nails are driven, although these facts could in theory be part of the design. In short, there is a certain threshold of detail that is agreed upon by the designer and implementor as being *primitively achievable* or *assumable* — any such specification need not be further elaborated. An adequate design (or simply "a design") can be thought of in these terms as *consistent set of primitively achievable specifications.*

2.2 Design by Finding Residues

Design synthesis has long been approached as a problem of extracting a design from a proof. W. S. Cooper's [18] 1964 system, and James Slagle's 1965 system DEDUCOM [84] were perhaps the first steps in that direction. In 1969, Cordell Green's QA3 [32] and Waldinger

[1] To avoid confusion, the *the object to be designed* and *the design specification itself* need to be distinguished. The use of the expression "speeding up the design," is also avoided as it is not clear whether the design process is accelerated or whether the the object created by the design will run faster in some sense.

[2] Another possibility is that the design output will be used as the specification for some other design problem. In VLSI design, for example, one might synthesize a sticks level design that is used as the input to another design problem, namely, layout. Having specified the level of the second design, the design problem remains the same, however.

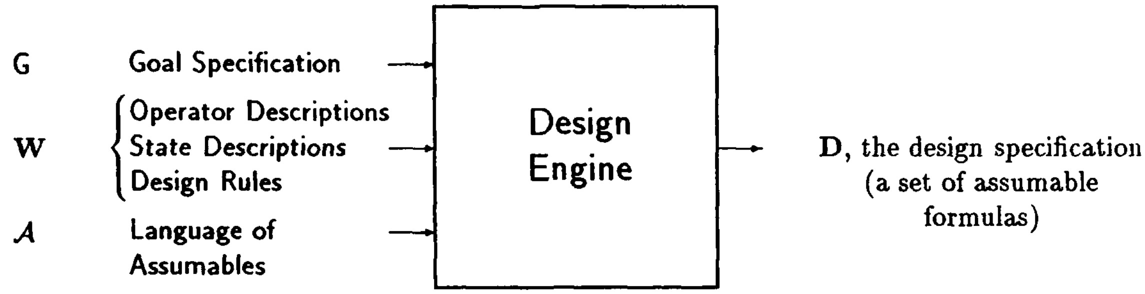


Figure 1: Design Synthesis: A Mapping from One Set of Specifications to Another

and Lee's PROW [99,98] synthesis systems were both published. QA3 and PROW were both resolution-based theorem-proving systems, formulating similar theorems to be proven, but quite different in their answer extraction techniques. The approach developed in these early systems (called the *single-term approach* in this thesis) has continued to be used to the present. In constrast, this section presents an alternative approach to deductive design called the *residue approach*.[3] As explained in Chapter 1, the residue approach starts with a goal G, world model **W**, and a language $\mathcal{A}$ of assumable specifications, and from these a design specification **D** must be found. The approach is illustrated in Figure 1 and its main components are described in the following subsections.

2.2.1 The Goal G

The object to be designed is described by a closed formula, G, called the *goal* or *design goal*. For now, it will be assumed that the values for any existentially qualified variables are not of interest. Later, in Section 2.6 it will be shown how values for existential variables can be extracted from the proof of a residue.

> **Example 2.1** In a blocks world, consider the goal that some block be on top of block B at some future time t_f. If the current time is T_i, then the goal might be expressed as the formula
>
> $$\mathsf{G} = \exists t_f\, \mathsf{True}(\mathsf{On}(\mathsf{x},\mathsf{B}),t_f) \wedge (t_f \geq \mathsf{T}_i).$$

[3]Chakravarthy [12,14] has used the term *residue* in a quite different manner as will be seen in Chapter 4.

Note that the above goal formula does not mention the design explicitly, but only says what must be true at a some time in the future.

2.2.2 The World Model W

In every design problem, it will be assumed that there is a consistent set **W** of closed formulas modeling the designer's knowledge of the world. In future sections, there will be differing assumptions made about the allowed forms of the formulas in **W**, e.g., that the formulas in **W** are implicitly quantified clauses or Horn clauses. In the literature, a distinction is sometimes made between *facts* (generally meaning ground atomic formulas) and *rules* (generally meaning conditionals, usually containing universally quantified variables). No such distinction is made here.

The axioms of **W** are assumed to represent a number of different sorts of information, the distinction between them being somewhat arbitrary:

- **Operator Descriptions:** axiomatization of the functionality of the components from which objects may be synthesized. This set of components need not be the most basic building blocks, but might be a hierarchy of possible components. In the blocks world, **W** *might contain* the the following rule to describe a puton action:

$$\begin{array}{ll} \forall x, y, t & \mathsf{True}(\mathsf{Clear}(x), t) \wedge \\ & \mathsf{True}(\mathsf{Clear}(y), t) \wedge \\ & x \neq y \wedge \\ & \mathsf{True}(\mathsf{Handempty}, t) \wedge \\ & \mathsf{Execution}(\mathsf{puton}(x, y), t) \supset \mathsf{True}(\mathsf{On}(x, y), t+1) \end{array}$$

Similarly, to describe one state of a NAND-gate x, **W** might contain:

$$\begin{array}{ll} \forall a, b, c, x, t & \mathsf{NAND\text{-}gate}(x) \wedge \\ & \mathsf{Input}(x, 1, a) \wedge \\ & \mathsf{Input}(x, 2, b) \wedge \\ & \mathsf{Output}(x, 1, c) \wedge \\ & \mathsf{True}(\mathsf{Value}(a, \text{“1”}), t) \wedge \\ & \mathsf{True}(\mathsf{Value}(b, \text{“1”}), t) \supset \mathsf{True}(\mathsf{Value}(c, \text{“0”}), t) \end{array}$$

- **State Description:** Facts about the state from which the object is to be constructed. In synthesis of an electric circuit, one is usually constructing a circuit from scratch, but for other synthesis problems, say a robot planning problem, a crucial part of the

problem is knowing the initial state, that is, the state in which the world will be when implementation of the design (i.e., execution of the plan) begins. In addition, if other relevant state transitions will occur independent of the execution of the design, these must also be described.

The following formula might express that block A is initially on the table:

$$\mathsf{True}(\mathsf{On}(A, \mathrm{Table}), 0),$$

or the fact that block B is fragile might be expressed as:

$$\forall x, t\ \mathsf{True}(\mathsf{On}(x, B), t) \supset \mathsf{True}(\mathsf{Broken}(B), t+1).$$

- **Design Rules:** Additional formulas may represent the allowed configurations of design components, for example, there might be a prohibition on two inputs to the same NAND-gate:

$$\begin{aligned} \forall a, b, i, x \quad & \mathsf{NAND\text{-}gate}(x) \wedge \\ & \mathsf{Input}(x, i, a) \wedge \\ & \mathsf{Input}(x, i, b) \supset \quad a = b \end{aligned}$$

2.2.3 Assumables

As discussed in Section 2.1, for a design to be useful, it must be sufficiently detailed so that the intended implementor is capable of carrying it out. In designing a house, "Bolt board A to board B" is a primitively achievable goal; we need not further specify this operation. In VLSI layout, we assume that as long as certain design rules are followed, *any* shape of conductor can be put into the silicon.[4] Again, there is no need for the task to be specified further.

For problems discussed here, *it will be assumed that a primitively achievable instruction is expressed as an atomic formula*, ground or otherwise. Such primitively achievable instructions will be called *assumables*. Furthermore it is assumed that there exists a language $\mathcal{A}$ of assumables and an algorithm that decides in negligible time whether or not a formula is in $\mathcal{A}$. In practice, the language of assumables will be specified by a set of relations and polarities (i.e., negated or not), such that any atomic formula with the appropriate polarity

[4] Whether the design works as intended is a function of its following the "design rules" for that fabrication process. It is assumed that such design rules are part of W and are guaranteed by the consistency requirement (discussed in Section 2.2.5.

and relation is assumable. As stated in Chapter 1, it will be convenient to assume that $\mathcal{A}$ is such that a conjunction $A_1 \wedge \ldots \wedge A_n$ is assumable if each of the A_i is assumable.

Example 2.2 In the blocks world planning problem, the following schemata might describe the language $\mathcal{A}$ of assumables:

$$\begin{array}{c}
\mathsf{Execution}(<\!\mathit{action}\!>,<\!\mathit{time}\!>) \\
<\!\mathit{time}\!>_1 \;<\; <\!\mathit{time}_2\!> \\
<\!\mathit{time}\!>_1 \;\leq\; <\!\mathit{time}_2\!>
\end{array}$$

2.2.4 Definition of Design

Given the above discussion of a goal G, a world model **W**, and a language $\mathcal{A}$ of assumable formulas, let us define a *residue* as follows:

Definition 2.1 (Residue) *Given a consistent set* **W** *of well-formed formulas, a closed formula* G, *and a set of well-formed formulas* $\mathbf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$, **D** *is a* legal design *if*

1. $\mathbf{W}, \mathbf{D} \models \mathsf{G}$ **(Sufficient)**,
2. $\mathbf{W} \cup \mathbf{D}$ *is satisfiable* **(Consistent)**, *and*
3. *For all* $\mathsf{D}_i \in \mathbf{D}$, $\mathsf{D}_i \in \mathcal{A}$. **(Assumable)**.

The requirement of *sufficiency* is straightforward — Given the world model **W**, the goal G must be entailed by the world model and the design. The requirement of *assumability* has been discussed in previous sections. Following an example, the requirement of *consistency* will be discussed in Section 2.2.5.

Consider a simple example of a design using only propositional calculus.

Example 2.3 Suppose the set of assumable formulas, facts of the world model **W**, and goal G are as below:

$$\begin{array}{ll}
\mathcal{A}: & A, B, C, D, E, F \\
\mathbf{W}: & B \wedge D \rightarrow J \\
 & D \wedge E \rightarrow K \\
 & J \wedge K \rightarrow M \\
\mathsf{G}: & M
\end{array}$$

Given the above and rules, $\mathbf{W} \cup \{B, D, E\} \models M$, as illustrated by the series of reduction steps below:

$$\begin{array}{c} M \\ | \\ J \wedge K \\ | \\ B \wedge D \wedge K \\ | \\ B \wedge D \wedge E \end{array}$$

The set of propositions $\{B, D, E\}$ is consistent with $\mathbf{W}$, that is,

$$\mathbf{W} \not\models \neg(B \wedge D \wedge E),$$

so $\{B, D, E\}$ meets all three criteria of Definition 2.1, and is a residue for the goal M given world model $\mathbf{W}$ and assumables $\mathcal{A}$.

2.2.5 Consistency of the Design

Let us assume that the axioms in $\mathbf{W}$ accurately describe some portion of the real world. Then, for some $\mathbf{D}$, if $\mathbf{W} \cup \mathbf{D}$ is unsatisfiable, one must assume that the D_i cannot simultaneously describe the any configuration of the real world. As a minimum condition for implementing a design, the *consistency* condition of Definition 2.1 must hold.

Example 2.4 Suppose $\mathbf{W}$ contains a rule saying that "No two actions can take place simultaneously," i.e.,

$$\begin{aligned} \forall a, b, c, x, t \quad & \mathsf{Execution}(a, t_1) \wedge \\ & \mathsf{Execution}(b, t_2) \supset \quad t_1 \neq t_2. \end{aligned}$$

By ignoring this rule, one might produce a plan for switching the positions of two blocks A and B simply by saying, "Move block A to the location of block B at time T", and "Move block B to the location of block A at time T," i.e.,

$$\mathsf{Execution}(\mathsf{Move}(\mathsf{A}, \mathsf{B}), T) \wedge \mathsf{Execution}(\mathsf{Move}(\mathsf{B}, \mathsf{A}), T).$$

Given the usual axiomatizations of the Move operator, the goal logically follows from a plan executing these two actions simultaneously, but there would be no way to implement this plan in the real world.

In the above example, a design that was impossible to implement was disallowed. The consistency requirement also acts to enforce design rules; if the design rules are not satisfied, there is no guarantee that the object designed will behave as expected. For example, it is possible to implement a VLSI layout that puts insufficent space between adjacent conductors, but it probably will not work in the manner the designer planned.

One might ask, "Is there a way for a synthesis system to avoid the need for consistency checking via careful axiomatization?" The answer is, "yes," but only in certain cases. This issue will be discussed further in Section 2.7.

The Complexity of Consistency Checking In everyday life, given some flaw in a design, we presume we can eventually find it (given enough time and assuming we are capable of understanding the flaw and its causes), *but we cannot say that how long it will take to find the flaw.* In checking whether a design is consistent, the situation is exactly analogous. Determining the satisfiability of a set of first-order formulas is a non-semidecidable problem. In other words, there is no procedure that can take an arbitrary set of formulas and always determine in a finite amount of time that the set of formulas is satisfiable. On the other hand, showing that a set of formulas is unsatisfiable is semidecidable, that is, there are procedures guaranteed to prove an unsatisfiable set to be unsatisfiable in a finite amount of time. So, if a set of first-order formulas is inconsistent,[5] we can eventually discover this fact, but there is no way, in general, to ever be sure that a given set of facts is consistent.

The impact of the above phenomenon for a design problem is clear: either the language of **W**, G and **D** must be restricted, or one must settle for less than perfect guarantees of consistency. There are numerous useful subsets of first-order logic that are decidable, that is, for which there exists an algorithm for deciding in a bounded amount of time whether or not a given formula is valid (and as a result, whether a given set of formulas is consistent). Besides the obvious example of the propositional calculus, Manna [57], page 107, gives many examples of other decidable subsets of the first order predicate calculus.

Non-Guaranteed Consistency Checks Although checking consistency of a set of facts in intractable, for a given problem at the least one hopes to find some way to do an acceptable, though imperfect job of checking that a design will behave as expected.

For a given **W** and **D** it is possible that all search paths will be exhausted in trying to show that $\mathbf{W} \models \neg \mathsf{D}$, where D is the conjunction of all $\mathsf{D}_i \in \mathbf{D}$. In such a case, the design **D** is consistent with **W**. Failing this, one would like to assume that given an inconsistent

[5] ***Inconsistency*** and ***unsatisfiability*** are equivalent conditions in first-order logic, as was proven by Gödel in 1930.

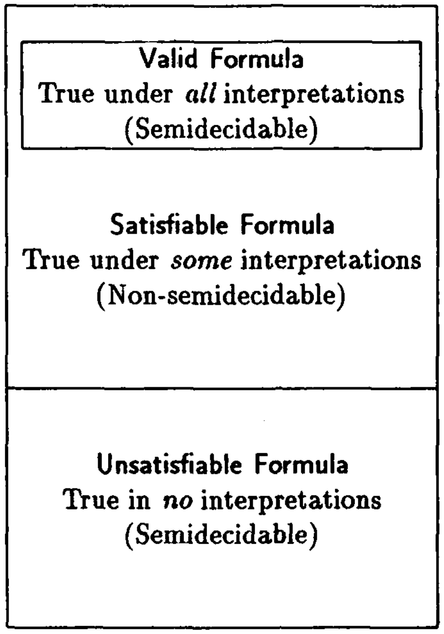


Figure 2: Valid, Satisfiable, and Unsatisfiable Formulas of First-Order Logic

design **D**, the likelihood of proving the design inconsistent increases monotonically with the amount of time spent in checking consistency. In other words, one might want to assume that if no inconsistency is found after spending one minute of cpu time, it is quite unlikely that the design is inconsistent; if after two minutes inconsistency has not been proven, it is even less likely that the design is inconsistent.

Given the above assumption, a residue procedure can legislate the amount of time or space spent looking for inconsistency before declaring the design to be consistent. Such a decision might also be based on the course of the proof, i.e., whether or not it looks "promising" that the design will be proven inconsistent, although that possibility will not be considered further here.

Finally, for a given problem it might be reasonable to rely on some ad hoc method of consistency checking. One may choose to enumerate the classes of potential problems and check only these possibilities. The checking can be via arbitrary procedures rather than by any logical inference. The "critics" of Sussman's HACKER [94] or Sacerdoti's NOAH [81] are typical of such an approach.

Example 2.5 Consider building a combinational circuit out of a set of wires

and NAND-gates. There are a limited number of ways the system can go wrong, i.e., that is, fail to act as expected in spite of the correct wires being in place:

1. There may be a loop in the circuit.
2. There may be more than one output connected to a single input.

The above two design rules can quickly be checked, and assuming that they are not violated, the designer may be willing to assume (or "legislate") that the circuit will work as expected.

2.3 Residue Procedures for Design Synthesis

The previous section formulated design synthesis as the problem of finding residues for a goal G, a world model **W**, and a language of assumables $\mathcal{A}$. The current section and the two following describe two procedures for finding residues. This section describes the notions common to the entire class of what might be called "residue procedures." Section 2.4 presents Ordered Residue, a residue procedure based upon backwards inference very similar to that done in MYCIN [82,9] and Prolog [78]. Ordered Residue will be used extensively in later chapters. Section 2.5 presents Resolution Residue, a more general residue technique using binary resolution [76] as its backwards inference technique. Appropriate completeness results are proven for each.

Figure 3 is a high level description of a canonical residue procedure. The central idea is that the original goal G is reduced to other goals[6] via a sequence of goal reduction[7] steps until an assumable goal is found, that is, a goal for which $D \in \mathcal{A}$. In Figure 3, the reduced goal is represented by the symbol D to emphasize the fact that any goal is potentially a proposed design. If, at any time, the reduced goal/design is inconsistent (that is, $\mathbf{W} \models \neg D$), then this path can be pruned; further reduction steps cannot make the inconsistent design consistent.

The step "D ← Goal Reduction(D)" in Figure 3 is nondeterministic. On each iteration through the loop, the goal reduction step may produce none, one or many new goals D.

[6]It is tempting to say, "the goal G is reduced to various subgoals," but the term "subgoal" has a very specific (and different) meaning in logic programming, namely, a goal is a conjunction of literals and a subgoal is one of the conjuncts. To avoid confusion, the word "subgoal" will usually be avoided here. In the rare instances in which the word "subgoal" appears, it will refer to the entire goal to which another goal has been reduced.

[7]*Goal reduction* known by many other names in the literature, among them *backwards inference*, *goal-directed reasoning*, *subgoaling*, *top-down reasoning*, *goal regression*, and *consequent reasoning*. Here, the terms "goal reduction" and "backwards inference" will be used interchangably.

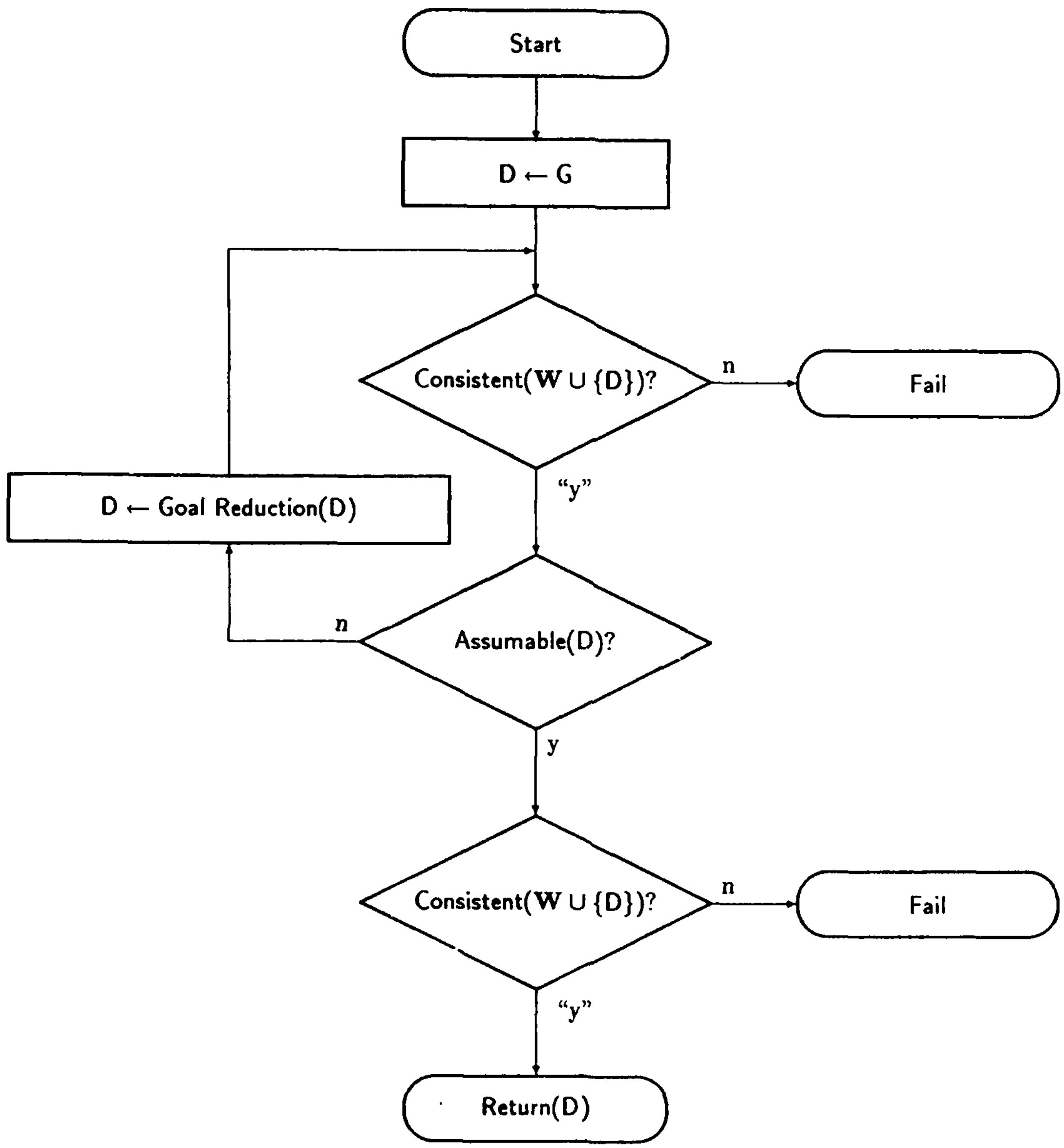


Figure 3: Simplified View of a Residue Procedure

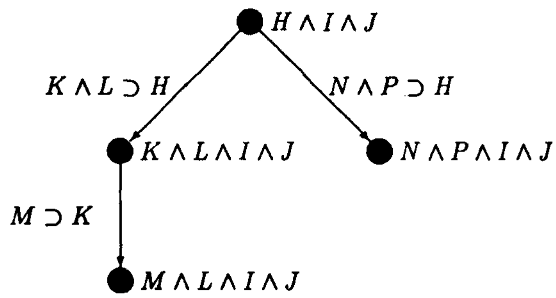


Figure 4: Goal Reduction Steps

Each of the new goals D is a new path in the space of possible goal reductions from the original goal G. If for some D no goal reductions are possible, that path simply halts without returning an answer.

> **Example 2.6** Consider the goal $\mathsf{G} = H \wedge I \wedge J$, where **W** contains the axioms $K \wedge L \supset H$, $M \supset K$, and $N \wedge P \supset H$. Figure 4 shows possible goal reductions that might be made. Each node shown represents a point in the space searched by the nondeterministic step "D $\leftarrow$ Goal Reduction(D)."

The "Consistent(D)?" steps in the flowcharts describe a program step solving an intractable problem, an obvious impossibility. "Consistent(D)?" should be understood to refer to the heuristic approach (described in the previous section) being used in the particular problem to approximate deciding whether $\mathbf{W} \models \neg \mathsf{D}$. Note that the consistency checking step appearing in the loop is not needed for correctness of the procedure, but only to prune goals that are inconsistent as soon as possible. In practice, one may chose to eliminate this step entirely or make it a minimal sort of check.

For simplicity's sake another fiction that has been allowed in Figure 3. Every reduced goal in the figure springs from exactly one previously existing goal. It is possible, however, to use more than one goal $\mathsf{D}_1, \ldots, \mathsf{D}_n$ to find a new goal D' such that $\mathbf{W} \cup \{\mathsf{D}\} \models \mathsf{G}$. Such is the case in Resolution Residue, but the distinction is not crucial here.

The rest of this chapter concerns itself with the goal reduction component of residue procedures and is independent of the consistency checking. In future chapters, goal reduction (as opposed to consistency checking) will sometimes be called the "backwards component" of the residue procedure because it consists of backwards inference from a goal description. Consistency checking will sometimes be referred to as the "forward component" of

the procedure, since one can think of consistency checking as forward inference from the set $\mathbf{W} \cup \{\mathsf{D}\}$ — if **false** is derived from $\mathbf{W} \cup \{\mathsf{D}\}$, then D is not a consistent design. As can be seen from Figure 3, the forward and backwards components compete for resources. Chapters 3 and 4 will generalize consistency checking and deal with the competition between the forward and backwards components.

2.4 Ordered Residue

This section presents the Ordered Residue procedure. In a nutshell, Ordered Residue is the residue procedure obtained by performing backwards inference via Ordered Resolution on Head-First Ordered Horn Clauses under the set of support restriction. All of these terms will be defined later in this section. In addition, Ordered Resolution under the above restrictions is a close relative of Prolog [78] as will be discussed in Section 2.4.4.

2.4.1 Ordered Resolution

In the literature on resolution, clauses are variously defined as sets or bags of literals, and they may be ordered or unordered. For Ordered Resolution,[8] a clause is a sequence of distinct literals, that is, an ordered set. Definitions 2.2-2.4 are from Chang and Lee [15].

Definition 2.2 (Ordered Clause) *An* ordered clause *is a sequence of distinct literals.*

Definition 2.3 *A literal L_2 is said to be* greater than *a literal L_1 in an ordered clause (or L_1 is* smaller than *L_2) if and only if L_2 follows L_1 in the sequence specified by the ordered clause.*

Ordered Resolution is defined with Definitions 2.4–2.6.

Definition 2.4 (Ordered Factor) *If two or more literals (with the same sign) of an ordered clause C have a most general unifier σ, then the ordered clause obtained from the sequence $C\sigma$ by deleting any literal that is identical to a smaller literal in the sequence is called an* ordered factor *of C.*

Definition 2.5 (Ordered Binary Resolvent) *Let C_1 and C_2 be ordered clauses with no variables in common. Let L_1 and L_2 be the smallest literals in C_1 and C_2, respectively. If L_1 and $\neg L_2$ have a most general unifier σ, and if C is the ordered clause obtained by*

[8] The resolution rule defined here is a variant of Boyer's *Lock Resolution* [5], and it differs significantly from *Ordered Resolution* as defined in Chang and Lee [15], page 113.

concatenating the sequences $C_1\sigma$ and $C_2\sigma$, removing $L_1\sigma$ and $L_2\sigma$, and deleting any literal that is identical to a smaller literal in the remaining sequence, then C is called an ordered binary resolvent *of C_1 against C_2. The literals L_1 and L_2 are the* literals resolved upon.

Definition 2.6 (Ordered Deduction) *A clause C is said* to be deduced *via an* ordered deduction *from base set S of ordered clauses if and only if there is a tree T such that C is in T, every node in the fringe of T is a member of S, and for every other node D either:*

1. *D has one parent P, and D is an ordered factor of P, or*
2. *D has two parents P and Q, and D is an ordered binary resolvent of P against Q.*

Soundness of Ordered Resolution follows from the set of allowed steps being a subset of the allowed steps in ordinary Binary Resolution. Treitel and Genesereth [95] have proved the following completeness theorem:

Theorem 2.1 (Completeness of Ordered Resolution on Horn Clauses) [Treitel and Genesereth] *A set S of ordered Horn clauses is unsatisfiable is and only if there is an ordered deduction of the empty ordered clause $\Box$ from S.*

Ordered Residue uses the set of support restriction, defined as follows:

Definition 2.7 (Set of Support) *A subset T of a set S of clauses is called a* set of support *of S if $S - T$ is satisfiable. An (ordered)* set of support resolution *is an (ordered) resolution of two clauses that are not both from $S - T$. An (ordered)* set of support deduction *is a deduction in which every (ordered) resolution step is an (ordered) set of support resolution.*

Unless otherwise stated, the initial set of support T is understood to be the set of clauses from the negation of the goal.

Ordered resolution is complete for Horn clauses, but unfortunately it is not compatible with the set of support restriction unless the positive literal is always the smallest (first) literal in the clause. Consider the following example:

Example 2.7 Suppose we have a database

$$\{B \supset A, B\}$$

and a goal A. This would correspond to the unsatisfiable set of ordered clauses

$$\{\neg A, \neg B \vee A, B\},$$

and initial set of support $\{\neg A\}$. Ordered resolution would not be able to prove this set unsatisfiable under the set of support restriction since the ordered clause $\neg B \vee A$ does not resolve with $\neg A$. If the ordered clause $\neg B \vee A$ were replaced with the ordered clause $A \vee \neg B$, then there would be an ordered refutation for this set.

(although an ordered clause $A \vee \neg B$ would succeed since it can resolve with $\neg A$).

We give the name HOH-clause to Horn clauses with the first literal being the positive one.

Definition 2.8 (HOH-Clause) *A* Head-first Ordered Horn clause *(HOH-clause) is an ordered clause such that all literals in the sequence are negative with the possible exception of the first, which may be positive.*

Given a formula such as $B \wedge C \wedge D \supset A$, any (or all) of the HOH-clauses below might be found in the database:

$$A \vee \neg B \vee \neg C \vee \neg D$$
$$A \vee \neg B \vee \neg D \vee \neg C$$
$$A \vee \neg C \vee \neg B \vee \neg D$$
$$A \vee \neg C \vee \neg D \vee \neg B$$
$$A \vee \neg D \vee \neg C \vee \neg B$$
$$A \vee \neg D \vee \neg B \vee \neg C$$

For Ordered Resolution on HOH-clauses, the set of support restriction preserves completeness, as has been proven by Treitel and Genesereth [95]:

Theorem 2.2 [Treitel and Genesereth] *Given a goal* G *such that the clauses of* $\neg$G *contains only negative literals, a database* **W** *of HOH-clauses, and using the ordered clauses from* $\neg$G *as the initial set of support, there exists an ordered deduction of the null clause* $\Box$ *from base set* $\mathbf{W} \cup$ Clauses(G) *and initial set of support* Clauses(G) *if and only if* $\mathbf{W} \models$ G.

2.4.2 Ordered Residue Procedure

In order to use ordered resolution for generating residues, it is necessary to somehow notice negated assumables as the first literal and move them so that the assumable does not prevent other literals in the clause from being used in subsequent ordered resolution

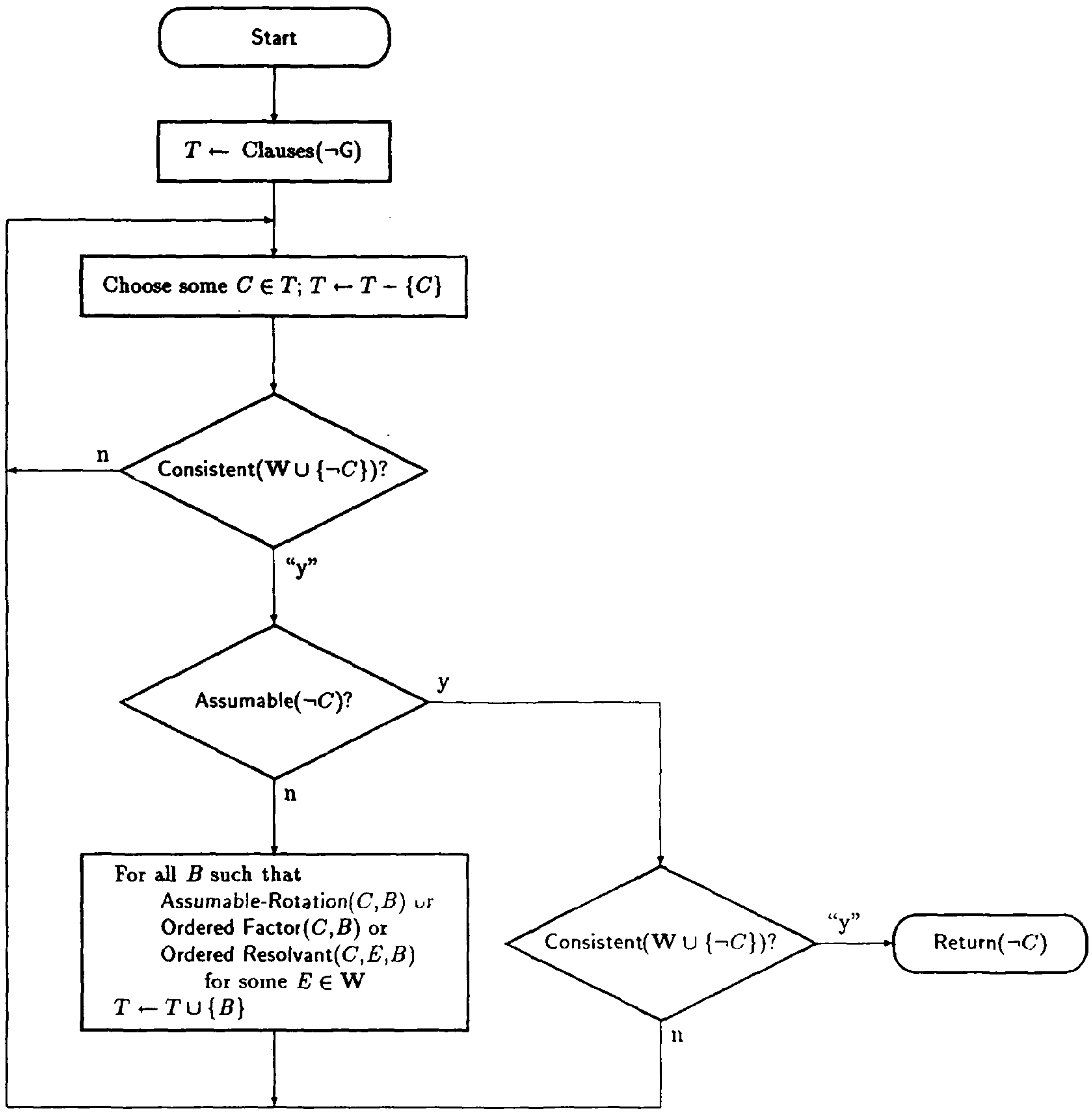


Figure 5: The Ordered Residue Procedure

steps. One simple way to accomplish this goal is to allow the rotation of a negated assumable from the first literal to being the last literal. Let us define a relation called Assumable-Rotation(a, b) as follows:

Definition 2.9 *A pair of ordered clauses a and b is a member of the* Assumable-Rotation *relation if and only if for ordered clause* $a = a_1 \vee \ldots \vee a_m$, $\neg a_1$ *is an assumable, and* $b = a_2 \vee \ldots \vee a_m \vee a_1$.

In other words, b has rotated the first literal of a to the rear. Such a rotation can not change completeness of Ordered Resolution since it only adds to the range of possibilities. Such a rotation also preserves soundness, as the meaning of a derived clause is not a function of the clause's order. The Ordered Residue Procedure may then be defined as in Figure 5.

The "Choose some $C \in T$" step has not been fully specified. The implementation of this step controls the search, e.g., by always choosing the *most* recently added member of T, one obtains a depth-first behavior. Similarly, by always choosing the *least* recently added member of T, one obtains a breadth-first search. Note also that the clause C chosen is removed from T after being chosen; there is no further need for it.

2.4.3 Completeness of Ordered Residue

The completeness results of this section and of Section 2.5.3 are expressed in terms of one clause subsuming another. The definition of subsumption is the same for ordered and ordinary clauses, treating them both as *sets* of literals.

Definition 2.10 *A clause* C subsumes *a clause* D *if and only if there is a substitution* σ *such that* $C\sigma \subseteq D$. D *is called a* subsumed *clause.*

The main result of this section can now be stated:

Theorem 2.3 (Completeness of Ordered Residue on HOH-clauses)
Given a residue $\mathbf{D}$ *for world model* $\mathbf{W}$, *goal* G, *and assumable language* $\mathcal{A}$, *where*

$\mathbf{W}$ *is a satisfiable set of HOH-clauses,*

$\mathsf{G} = \mathsf{G}_1 \vee \ldots \vee \mathsf{G}_m$, *where the* G_i *are conjunctions of positive literals, and*

$\mathbf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ *is a set of atomic assumables,*

there exists an ordered set of support deduction of a clause $A = \neg a_1 \vee \ldots \vee \neg a_q$ *from initial set of support* Clauses$(\neg\mathsf{G})$ *such that* $\{a_1, \ldots, a_q\}$ *is a residue for* $\mathbf{W}$, G, *and* $\mathcal{A}$ *and such that* A *subsumes* $\neg\mathsf{D}_1 \vee \ldots \vee \neg\mathsf{D}_n$.

Before proving Theorem 2.3, a number of other results will be helpful. The following "lifting lemma" (as stated in Wos et al [106]) is the appropriate formulation for a resolution systems with separate resolution and factoring steps. It differs from the common formulation found in Chang and Lee [15].

Lemma 2.4 (Lifting Lemma) [Robinson] *If A' and B' are, respectively, ground instances of (ordered) clauses A and B (which are assumed to have no variables in common), and if C' is a (ordered) resolvent of A' and B', then there exist (ordered) clauses E and F such that an application of (ordered) binary resolution to E and F yields a clause C, where C' is an instance of C, and where $E = A$ or is a factor of A and $F = B$ or is a factor of B.*

In addition, Herbrand's Theorem will be used:

Lemma 2.5 [Herbrand] *A set S of (ordered) clauses is unsatisfiable if and only if there exists a finite set of ground instances of S that is truth-functionally unsatisfiable.*

Finally, the following lemma will be useful.

Lemma 2.6 *For any formulas G and A, if there exists a derivation of a clause $\neg\mathsf{A}$ from base set $\mathbf{W} \cup \text{Clauses}(\neg\mathsf{G})$, then $\mathbf{W}, \mathsf{A} \models \exists\mathsf{G}$.*

> **Proof:** Let us define $\mathsf{G}_0 = \mathsf{G}_1 \vee \ldots \vee \mathsf{G}_m$, where $\text{Clauses}(\neg\mathsf{G}) = \{\neg\mathsf{G}_1, \ldots, \neg\mathsf{G}_m\}$. By the soundness of resolution, the derivation of $\neg\mathsf{A}$ means that
>
> $$\mathbf{W}, \neg\mathsf{G}_0 \models \neg\mathsf{A}. \tag{2}$$
>
> Via the deduction theorem and a series of equiavalences we get
>
> $$\begin{aligned} \mathbf{W} &\models \neg\mathsf{G}_0 \supset \neg\mathsf{A} && (3)\\ \mathbf{W} &\models (\exists\mathsf{A}) \supset (\exists\mathsf{G}_0) && (4)\\ \mathbf{W}, \exists\mathsf{A} &\models \exists\mathsf{G}_0 && (5)\\ \mathbf{W}, \mathsf{A} &\models \exists(\mathsf{G}_1 \vee \ldots \vee \mathsf{G}_m). && (6) \end{aligned}$$
>
> Formula (6) is the desired formula. It says that there exists some solution to G for any values of the free variables of A. ∎

Proof of Theorem 2.3 **Case I:** (W $\models \exists$G)

The theorem is vacuously true for this case. Let **D** be the null set. Then we have

1. **D** subsumes {}
2. **W** $\cup$ **D** is satisfiable.
3. **W** $\cup$ **D** $\models \exists$G.

Case II: (W $\not\models \exists$G)

Let **M** = **W** $\cup$ **D** $\cup$ Clauses($\neg$G). Since **W** $\cup$ **D** $\models$ G, **M** is unsatisfiable. By Herbrand's Theorem (Lemma 2.5), there must exist some unsatisifiable set $\mathbf{M}_0$ of ground instances of **M**. Let $\mathbf{M}_W$ be the subset of clauses in $\mathbf{M}_0$ from **W**, $\mathbf{M}_D$ be the subset of clauses in $\mathbf{M}_0$ from **D**, and $\mathbf{M}_G$ be the subset of clauses in $\mathbf{M}_0$ from Clauses($\neg$G). Since **W** $\cup$ **D** is satisfiable, $\mathbf{M}_W \cup \mathbf{M}_D$ is satisfiable and $\mathbf{M}_G$ is non-empty. By Theorem 2.2 there exists an ordered set of support refutation R_0 of $\mathbf{M}_0$ using $\mathbf{M}_G$ as the initial set of support. Since W $\not\models \exists$G, R_0 contains at least one clause from $\mathbf{M}_D$ in its fringe.

Consider now the following two lemmas:

Lemma 2.7 *Given two ground ordered clauses A_1' and A_2' for which C' is an ordered binary resolvent, then there exists an ordered binary resolvent C for ordered clauses A_1 and A_2, where A_1 is any ordered clause consisting of the same sequence of literals as A_1' with some (possibly empty) set Z_1 of ground literals anywhere interspersed in its sequence except the first position, and A_2 is any ordered clause consisting of the same sequence of literals as A_2' with some (possibly empty) set Z_2 of ground literals anywhere interspersed in its sequence except the first position. Furthermore C consists of the same sequence of literals as C' with some (possibly improper) subset of $Z_1 \cup Z_2$ interspersed in its sequence.*

Proof: Since the first literals of A_1' and A_2' are unchanged in A_1 and A_2, A_1 and A_2 can be resolved as were A_1' and A_2'. The resulting ordered clause C will be identical to C' except for the additional negated literals introduced by A_1' and A_2'. Furthermore, $C - C' \subseteq Z_1 \cup Z_2$ since some of the literals introduced by Z_1 or Z_2 may be deleted if an identical and smaller literal is present.

Lemma 2.8 *Given a ground ordered clause A' for which C' is an ordered factor, then there exists an ordered factor C of ordered clause A, where A consists of the same sequence of literals as A' with some (possibly empty) set Z of negated ground literals interspersed in its sequence, and C consists of the same sequence of literals as C' with some (possibly improper) subset of Z interspersed in its sequence.*

Proof: For ground ordered clauses, factoring consists only of deleting duplicate literals. The literals of additional literals from Z can not affect a preexisting possible factorization. ∎

Consider some $a \in \mathbf{M}_D$, that is, a ground atomic assumable appearing in R_0. R_0 can be modified by eliminating an ordered resolution of a set of support ordered clause $\neg a \vee B$ against a ground assumable a. Instead of using the resolvent B, by Lemmas 2.7 and 2.8 a corresponding ordered set of support derivation R_1 can be constructed with the negated assumable $\neg a$ left in the tree and allowed to percolate toward the top. To build the tree, note that it usually will be necessary to rotate negated ground assumables such as $\neg a$ to the end of ordered clauses. Taking R_1, one can repeat the process constructing $R_2, \ldots, R_p$ until all members of $\mathbf{M}_D$ have been eliminated from the fringe of ordered set of support deduction R_p. R_p is then an ordered deduction of some clause C_p from base set $\mathbf{M}_W \cup \mathbf{M}_D$ such that (1) R_p consists only of negations of ground atomic assumables from $\mathbf{M}_D$, and (2) R_p is a set of support deduction with initial set of support $\mathbf{M}_G$.

By the Lifting Lemma (Lemma 2.4), R_p can be converted to another set of support derivation D_p^* such that the fringe consists of ordered clauses from $\mathbf{W} \cup \text{Clauses}(\neg\mathsf{G})$. The root of D_p^* is given by $C_p\theta^{-1} = \neg a_1' \vee \ldots \neg a_q'$ for some substitution θ, where the a_i' are atomic formula from $\mathbf{D}$ with some of the constants in these formulas possibly replaced by variables. Thus for each a_i', there exists some substitution ϕ such that $a_i'\phi \in \mathbf{D}$.
So,

1. $\neg a_1' \vee \ldots \neg a_q'$ subsumes $\neg \mathsf{D}_1 \vee \ldots \neg \mathsf{D}_n$
2. $\mathbf{W} \cup \{a_1', \ldots, a_q'\}$ is satisfiable since it has a satisfiable instance $\mathbf{M}_W \cup \mathbf{M}_D$
3. $\mathbf{W} \cup \{a_1', \ldots, a_q'\} \models \exists \mathsf{G}$ by Lemma 2.6.

∎

2.4.4 Relation to Prolog

As stated earlier, the set of steps allowed by Ordered Resolution is closely related to the set of steps made by a Prolog Interpreter [78]. Looking at the search space of Prolog as an AND-OR tree, with *conjunctions* of literals to solve via *disjunctions* of possible ways to reduce the literals, Prolog considers both conjunctions and disjunctions in depth-first fashion. In other words, given a goal (that is, a conjunction to be solved)

$$A \wedge B \wedge C,$$

Prolog will completely solve (either find an answer for or else fail on) A before considering B. Given rules (that is, a disjunction of possible ways to solve the conjunct A)

$$
\begin{aligned}
A &:- E, F, G \\
A &:- H, I, J, K \\
A &:- L, M, N,
\end{aligned}
$$

Prolog will also exhaust possibilities for finding answers to A via the first of the rules before considering the second. Thus, given a goal and an ordering of the rules in the database, Prolog specifies precisely the order in which inference steps may be made.

Ordered Resolution is similar to Prolog in handling goal conjuncts in order, but differs from Prolog in not specifying the order in which various rules may be applied. It will turn out that this depth-first ordering on conjuncts is important to the techniques of Chapters 3 and 4, but there is no need in this work to specify the order in which possible goal reductions on the same conjunct are tried.

2.5 Resolution Residue

Of the many inference techniques one might use as the backwards inference engine of residue, perhaps the most obvious one is binary resolution (Robinson [77]). Binary resolution (usually just called "resolution" for simplicity) has been greatly studied for two decades and is very well understood. In addition, it is easily implemented, and is refutation complete. To use resolution as a *backwards* inference technique, the set of allowed resolutions must be restricted since unrestricted resolution allows many more inferences than just backwards inference steps. Fortunately, one of the best known restrictions on resolution, the *set of support* restriction (Wos et al [105]), is exactly that — a restriction on resolution allowing only backwards inference steps. As shown by Wos, et al, the support restriction preserves refutation completeness. We might informally say that this results means that that one can walk a given search path from the goal to the initial state just as well as from the initial to the goal state.

The remainder of this section explores using resolution under the set of support restriction as the sole backwards inference technique in residue. The procedure so derived will be called *Resolution Residue* and is illustrated in Figure 6.

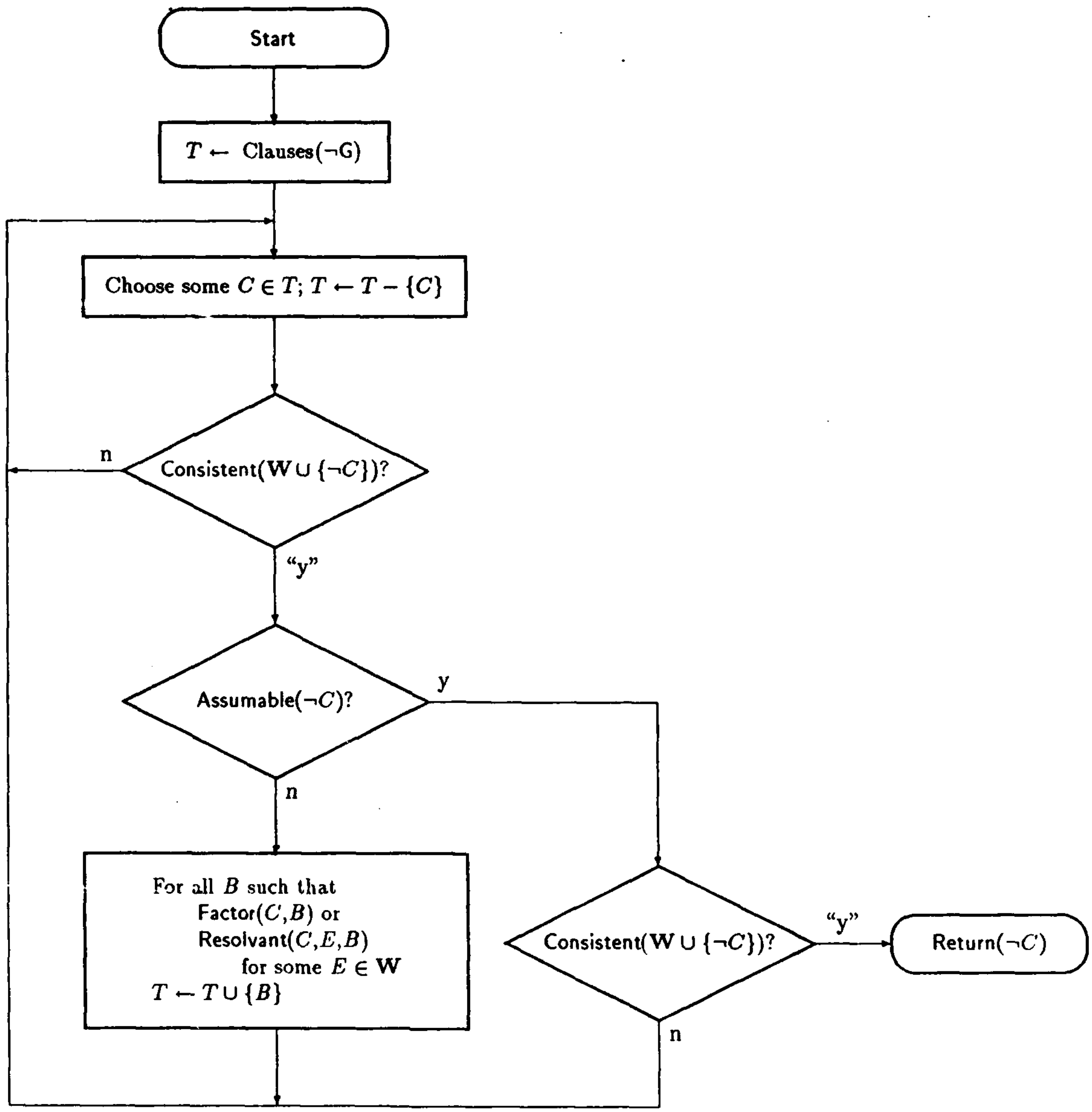


Figure 6: The Resolution Residue Procedure

2.5.1 Definitions

The definitions below are one standard formulation for binary resolution, differing from Robinson's original formulation in having separate factoring and resolution steps.[9] As contrast to Ordered Resolution, clauses are treated here as *sets* of literals rather than sequences. Somewhat different presentations of resolution and the set of support restriction may be found in texts by Chang and Lee [15], Manna [57], and Wos, et al [106].

Definition 2.11 (Factor) *If two or more literals (with the same sign) of a clause C have a most general unifier σ, then $C\sigma$ is called a* factor *of C. If $C\sigma$ is a unit clause, it is a called a* unit factor *of C.*

Definition 2.12 (Binary Resolvent) *Let C_1 and C_2 be two clauses (called* parent clauses*) with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2, respectively. If L_1 and $\neg L_2$ have a most general unifier σ, then the clause*

$$(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$$

is called a binary resolvent *of C_1 and C_2. The literals L_1 and L_2 are called the* literals resolved upon.

Binary resolvents usually are simply called *resolvents.*

Definition 2.13 (Deduction) *Given a set S of clauses, a (resolution)* deduction *of C from S is a finite sequence $C_1, C_2, \ldots, C_k$ of clauses such that each C_i is either a clause in S, a binary resolvent of two clauses preceding C_i or a factor or a clause preceding C_i. A deduction of the null clause $\Box$ from S is called a* refutation *of S.*

Theorem 2.9 states Robinson's well-known result that resolution is *sound and refutation complete.* It is given here without proof.[10]

Theorem 2.9 [Robinson] (Completeness of the Resolution Principle) *A set S of clauses is unsatisfiable if and only if there is a deduction of the empty clause $\Box$ from S.*

[9] The definitions here are similar to Loveland's [52] formulation of resolution.

[10] Because the formulation of resolution differs from Robinson, one must be careful in claiming his soundness and completeness. The formulation here defines factors and binary resolvants identically to Chang and Lee, but differs from Chang and Lee's definition of deduction in allowing separate factoring steps rather than only combined factoring-resolution steps. If one is willing to rely on the completeness result proven in Chang and Lee, it is clear that allowing a superset of Chang and Lee's steps cannot hurt completeness. In addition, since factoring steps are sound, soundness is preserved as well. Actually, the formulation of resolution here is identical to that of Loveland [52] in which clauses are sets of literals, and resolution and factoring steps are separate.

Theorem 2.10 is an important result from Wos et al [105], that of the completeness of the set of support strategy (Definition 2.7):[11]

Theorem 2.10 [Wos] (Completeness of the Set of Support Strategy) *If S is an unsatisfiable set of clauses and $T \subseteq S$ such that $S - T$ is satisfiable, then there exists a refutation of S with set of support T.*

2.5.2 The Resolution Residue Procedure

Figure 6 gives the procedure for Resolution Residue, with Resolvent(C, D, E) meaning that E is a binary resolvent of clauses C and D according to Definition 2.12 and Factor(C, E) meaning that E is a factor of clause C according to Definition 2.11. As illustrated in Figure 6, Resolution Residue follows the set of support restriction on an initial set of support of clauses from $\neg$G.

2.5.3 Completeness of Resolution Residue

This section closely follows the proof of completeness of Ordered Residue on HOH-clauses in Section 2.4.3. The completeness theorem for resolution residue is as follows:

Theorem 2.11 (Completeness of Resolution Residue)
Given a residue **D** *for world model* **W**, *goal* G, *and assumable language* $\mathcal{A}$, *where*

W *is a satisfiable set of clauses,*

$\mathsf{G} = \mathsf{G}_1 \vee \ldots \vee \mathsf{G}_m$, *where the* G_i *are conjunctions of literals, and*

$\mathbf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ *is a set of atomic assumables,*

there exists a set of support deduction of a clause $A = \neg a_1 \vee \ldots \vee \neg a_q$ *from initial set of support* Clauses($\neg$G) *such that* $\{a_1, \ldots, a_q\}$ *is a residue for* **W**, G, *and* $\mathcal{A}$ *and such that* A *subsumes* $\neg\mathsf{D}_1 \vee \ldots \vee \neg\mathsf{D}_n$.

Proof **Case I:** ($\mathbf{W} \models \exists$G)

The theorem is vacuously true for this case. Let **D** be the null set. Then we have

1. **D** subsumes {}
2. $\mathbf{W} \cup \mathbf{D}$ is satisfiable.
3. $\mathbf{W} \cup \mathbf{D} \models \exists$G.

[11] The wording here is from Loveland [52].

Case II: (W $\not\models \exists$G)

Let M = **W** ∪ **D** ∪ Clauses(¬G). Since **W** ∪ **D** $\models$ G, M is unsatisfiable. By Herbrand's Theorem (Lemma 2.5), there must exist some unsatisifiable set M_0 of ground instances of M. Let M_W be the subset of clauses in M_0 from **W**, M_D be the subset of clauses in M_0 from **D**, and M_G be the subset of clauses in M_0 from Clauses(¬G). Since **W** ∪ **D** is satisfiable, $M_W \cup M_D$ is satisfiable and M_G is non-empty. By Theorem 2.10 there exists a set of support refutation R_0 of M_0 using M_G as the initial set of support. Since **W** $\not\models \exists$G, R_0 contains at least one clause from M_D in its fringe.

Consider now the following two lemmas:

Lemma 2.12 *If ground clause C' is a resolvent of ground clauses A_1' and A_2' and A_1 and A_2 are such that $A_1 = A_1' \cup Z_1$ and $A_2 = A_2' \cup Z_2$ for sets Z_1 and Z_2 of ground literals, then there exists a ground literal $C = C' \cup Z_1 \cup Z_2$ such that C is a resolvent of A_1 and A_2.*

> **Proof:** A_1 and A_2 can be resolved using the same literals as was used in the resolution of A_1' and A_2'. The resulting clause C will be contain every literal of C' and in addition, will contain any extra literals from Z_1 and Z_2. ∎

Lemma 2.13 *Given a ground clause C' that is a factor of ground clause A', then there exists a factor $C = C' \cup Z$ of ground clause $A = A' \cup Z$, where Z is some (possibly empty) set of ground literals.*

> **Proof:** For ground clauses, factoring consists only of deleting duplicate literals. The literals of additional literals from Z can not affect a preexisting possible factorization. ∎

Consider some $a \in M_D$, that is, a ground atomic assumable appearing in R_0. R_0 can be modified by eliminating a resolution of a set of support clause $\neg a \vee B$ against a ground assumable a. Instead of using the resolvent B, by Lemmas 2.12 and 2.13 a corresponding set of support derivation R_1 can be constructed with the negated assumable $\neg a$ left in the tree and allowed to percolate toward the top. Taking R_1, one can repeat the process constructing $R_2, \ldots, R_p$ until all members of M_D have been eliminated from the fringe of ordered set of support deduction R_p. R_p is then a deduction of some clause C_p from base set $M_W \cup M_D$ such that (1) R_p consists only of negations of ground atomic assumables from M_D, and (2) R_p is a set of support deduction with initial set of support M_G.

By the Lifting Lemma (Lemma 2.4), R_p can be converted to another set of support derivation D_p^* such that the fringe consists of clauses from **W** ∪ Clauses(¬G). The root

of D_p^* is given by $C_p\theta^{-1} = \neg a'_1 \vee \ldots \neg a'_q$ for some substitution θ, where the a'_i are atomic formula from **D** with some of the constants in these formulas possibly replaced by variables. Thus for each a'_i, there exists some substitution ϕ such that $a'_i\phi \in \mathbf{D}$.
So, we have

1. $\neg a'_1 \vee \ldots \neg a'_q$ subsumes $\neg \mathsf{D}_1 \vee \ldots \neg \mathsf{D}_n$
2. $\mathbf{W} \cup \{a'_1, \ldots, a'_q\}$ is satisfiable since it has a satisfiable instance $\mathbf{M}_W \cup \mathbf{M}_D$
3. $\mathbf{W} \cup \{a'_1, \ldots, a'_q\} \models \exists \mathsf{G}$ by Lemma 2.6.

∎

2.6 Residue with Answer Extraction

Given a residue **D** for a goal G, we know that the union $\mathbf{W} \cup \mathbf{D}$ is sufficient to entail G. But, if G contains existentially quantified variables, it is sometimes desirable to know for what value of these variables the goal has been proven.

> **Example 2.8** The goal "Put a block on top of block A" might be stated as the goal
> $$\mathsf{G} = \exists x, t\ \mathsf{True}(\mathsf{On}(x, \mathsf{A}), t) \wedge (t > \mathsf{T}_0).$$
> Given some world model **W** and residue **D**, one knows that *some* block can be placed on block A, but without knowing *which* block x or at what time t. Knowing the value of t, x, or both might be important.

It is not always the case that there is a single variable binding that can be given as the value of an existential variable. Suppose a database contains the axiom

$$\mathsf{Dog}(\mathsf{Fido}) \vee \mathsf{Dog}(\mathsf{Rover}),$$

one is given the query

$$\exists x\ \mathsf{Dog}(x).$$

It can be proven that there is a dog, but it is impossible to say more than "Either Fido is a dog or Rover is a dog," that is, $x = \mathsf{Fido} \vee x = \mathsf{Rover}$. Such disjunctive variable bindings are called *indefinite answers* [75] in the database literature.

Cordell Green's method of *answer literals* [30,31] is well known for extracting such answers from resolution refutations. For each clause of the negation of the goal, the literal $\mathsf{Ans}(x)$ is added, where x is an existentially quantified variable of G whose value is of interest.

If there exists a resolution refutation T of $\neg$G, then a corresponding deduction tree T' can be built starting with G $\supset$ Ans(x), that is, with the answer literal Ans(x) added to each clause in the negation of G. The root of T' is a clause

$$\mathsf{Ans}(a_1) \vee \ldots \vee \mathsf{Ans}(a_n),$$

where it can be shown that $\mathbf{W} \models$ G for x equal to at least one of the a_i.[12] If there is a definite binding for $x = a$ for which $\mathbf{W} \models$ G (as proven by refutation T), then a unit clause Ans(a) will be the root of T'.

David Luckham and Nils Nilsson [54] found a slight generalization of Green's method in which they substitute the tautology $C_i \vee \neg C_i$ for each clause C_i of $\neg$G. Building a deduction tree via the same set of resolution and factoring steps as in the refutation of $\neg$G, Luckham and Nilsson deduce a disjunction of one or more formulas $\neg C_i$, but with the appropriate variables filled in with values used in the refutation. Since each such expression $\neg C_i$ is an instance of the goal (or a disjunct of the goal), one can extract appropriate values for each variable of interest. Luckham and Nilsson show that the answers they obtain are at least as general as those of Green (in the sense of binding variables with as little restriction on their values as possible). In addition, for considering values for several variables, Luckham and Nilsson's method returns disjunctions of bindings for each of the variables. In other words, if existential variables x, y and z appear in G, Luckham and Nilsson's method extracts a disjunction of triples of values, $\{x_1, y_1, z_1\} \vee \ldots \vee \{x_n, y_n, z_n\}$ rather than a disjunction of values for each variable separately. See Nilsson [72] for a more detailed exposition of this method.

Given an answer extraction method such as Green's of Luckham and Nilsson's, we might ask whether or not such a method can be used to find values for goals to a residue procedure. The proofs will not be presented here, but neither Green's nor Luckham and Nilsson's answer extraction techniques depend upon the deduction being a refutation (that is, a deduction of the null clause $\Box$), and thus, either method is applicable to both Ordered Residue and Resolution Residue.

For Ordered Residue, all extracted answers will be definite as well. Indefinite answers in resolution proofs can occur only when two clauses in the set of support are resolved against each other. Since all clauses in the initial set of support have only negative literals, and resolution against HOH-clauses preserves this condition on the set of support, then no two clauses in the set of support can ever be resolved against each other.

[12]As first suggested by Waldinger [49], indefinite answers can be avoided by generating a conditional whenever two clauses that have the same answer literals are resolved. Waldinger and Lee's PROW [99] generated conditionals in this fashion, and Green later incorporated conditional generation in QA3.

2.7 Discussion

The previous sections of this chapter presented both an approach to design and two residue procedures for generating design descriptions. In this section the residue approach is compared and contrasted to previous work in deductive synthesis.

2.7.1 The Single-Term Approach

In 1963 John McCarthy [66] proposed the *situational calculus*, that is, the representation of actions as a mapping from a state to a state, or a *situation* to a situation in McCarthy's terminology. Within the next two years a system by W. S. Cooper [18] and James Slagle's DEDUCOM [84] were published, these being first attempts at answering database queries via a proof that the answer exists. The year 1969 saw a synthesis of these approaches in Cordell Green's QA3 [32,31] and Waldinger and Lee's PROW [99,98].

Both QA3 and PROW derived designs by representing the design as a *term* about which the design specification can be proven to hold. Their approach is called the *single-term approach* here, and most previous work in deductive synthesis falls into this category. In the single-term approach, the proof procedure binds the term to an expression from which the design can be ascertained. This expression might be a constant (that is, a known situation in which the design specification holds) or more likely, a function (that is, is a composition of situation to situation transformations by which the desired situation can be reached). Consider the following example:

Example 2.9 To find out how to get block A on top of block B, the system proposes the goal formula

$$\exists d \ \mathsf{ON}(\mathsf{A},\mathsf{B},d)$$

and proves that this formula follows from the axioms describing the world. The situation d for which $\mathsf{ON}(\mathsf{A},\mathsf{B},d)$ will be bound to a description of a plan to achieve such a goal. Let us assume that block A is clear and that block C is on top of block B in the initial situation S_0. Then, one situation d for which $\mathsf{ON}(\mathsf{A},\mathsf{B},d)$ holds is

$$d = \mathsf{PUTON}(\mathsf{A},\mathsf{B},\mathsf{PUTON}(\mathsf{C},\mathsf{Table},\mathsf{S}_0)),$$

where the PUTON action is described by:

$$\forall a,b,s \ \ \mathsf{CLEAR}(a,s) \wedge \mathsf{CLEAR}(b,s) \supset \mathsf{ON}(a,b,\mathsf{PUTON}(a,b,s)).$$

We can find the design from the binding of d, that is, one puts A on B in the situation attained by putting C on the Table in S_0.[13]

In the single-term approach the problem is two-fold: (1) proving that the goal formula follows from the axioms describing the world, and (2) constructing the "answer," i.e., extracting the binding of the situation d for which the goal formula holds. Just as for the residue approach, any number of different proof procedures might be used. A number of different methods have been developed for answer extraction. Green [32] originally proposed the idea of an *answer literal* (See Section 2.6), an idea that was generalized by Luckham and Nilsson [54] in 1971 and by Manna and Waldinger [58] in 1980.[14]

2.7.2 Problems of Expression

As outlined in the above section, the single-term approach represents designs as a single term, that is, a composition of transformations of one situation to another. For problems in which backtracking through a space of state transformations provides adequate performance, the single-term approach works quite well. For other problems, the need to specify the design as a set of state transformations causes difficulty; a finer-grained set of constraints is desirable for specification of a design. Residues, that is, sets of atomic formulas, provide a richer language of constraints. Let us look at some of the difficulties encountered in the single-term approach and how they are handled in the residue approach.

Implicit Linear Ordering A term, being a composition of functions, implies a single linear order of application of the transformations. This ordering has been used in various domains to specify the ordering of components in the design. For example, in planning, the innermost function specifies the first action taken, the next innermost function specifies the next action, and so on. Similarly, in program synthesis, the composition of functions specifies application of functions in a functional programming language such as LISP. In circuit design, each function represents the output of a circuit component (such as a resistor) whose arguments are in turn functions specifying the inputs to that component.

[13]This example is a rough approximation of Green's methodology; Green actually developed and used answer literals (Section 2.6) to provide indefinite (disjunctive) answers in situations where an answer provably exists, but it is impossible to say what the answer is. Even so, each potential answer is represented as a single term.

[14]On the surface, Manna and Waldinger's approach does not seem to extract the answer as the binding of a term, but their "Output Column" can be viewed as a means to more freely rewrite the design term. In addition, authors such as Wos et al [106] represent designs via a fixed number of terms, but this does not change the fundamental nature of the discussion to follow. The issue is not one or many terms, but the use of terms rather than formulas.

If one chooses to build a specification in strictly backwards or strictly forward order, the above approach presents no problem. Each intermediate specification can be specified by a composition of functions. On the other hand, one cannot easily insert components into the middle of such a sequence. An intermediate design such as

$$\mathsf{PUTON}(\mathsf{A},\mathsf{B},\mathsf{PUTON}(\mathsf{B},\mathsf{C},s))$$

does not leave room for specification anywhere except at the innermost position.

Furthermore, because first-order logic does not allow quantification over functions, one cannot "leave an empty space" in the sequence of compositions. The expression

$$\mathsf{PUTON}(\mathsf{A},\mathsf{B},f(a_1,\ldots,a_n,\mathsf{PUTON}(\mathsf{B},\mathsf{C},s)))$$

is not a term of first-order logic. (In addition, one has no idea of how many arguments the function f will have.)

Residues, on the other hand, specify constraints in no particular order. The presence (or lack thereof) of a particular component can be expressed independently of its temporal or physical location in the design. As a result, there is no difficulty in specifying partial orderings, or in inserting a component between two others.

Example 2.10 By assuming the formulas

$$\mathsf{Execution}(\mathsf{Puton}(\mathsf{A},\mathsf{B}),\mathsf{T}_1) \quad \text{and} \quad \mathsf{Execution}(\mathsf{Puton}(\mathsf{B},\mathsf{C}),\mathsf{T}_2)$$

a residue procedure can easily state that actions Puton(A,B) and a Puton(B,C) are in a plan without stating in what order they will be executed. On the other hand, the single-term approach has no obvious way to do the same; it must decide in what order the actions are to take place via the design term

$$\mathsf{Puton}(\mathsf{B},\mathsf{C},\mathsf{Puton}(\mathsf{A},\mathsf{B},\mathsf{S}_0))$$

or else the term

$$\mathsf{Puton}(\mathsf{A},\mathsf{B},\mathsf{Puton}(\mathsf{B},\mathsf{C},\mathsf{S}_0)).$$

Specification of Components via Functions The single-term approach relies upon a set of state transformation functions to specify a design. As mentioned in the previous paragraph, first-order logic does not permit quantification over functions, so the expression

$$\mathsf{PUTON}(\mathsf{A},\mathsf{B},f(a_1,\ldots,a_n,\mathsf{PUTON}(\mathsf{B},\mathsf{C},s)))$$

is not a term of first-order logic. Once a state transformation function has been added to a term, it cannot be changed — there is no way to leave the state transformation a variable about which other constraints are specified.

Expressed as a residue, design components are terms rather than functions. For example, the formula

$$\mathsf{Execution}(a, t)$$

might specify that action a is to be executed at time t. One can certainly quantify over the set of actions a (and t, too) and specify additional constraints upon a such as

$$\neg\mathsf{Moves\text{-}Blocks}(a).$$

Combining Constraints Residues specify designs via an implied conjunction of the set of formulas in the residue. Single terms, on the other hand, cannot depend upon the semantics of logical conjunction.

> **Example 2.11** Consider the problem of stating that action A should take place more than 3 but less than 8 seconds after action B. A residue procedure can state
>
> $$\mathsf{Execute}(\mathsf{A}, \mathsf{T}_1)$$
> $$\mathsf{Execute}(\mathsf{B}, \mathsf{T}_2)$$
> $$\mathsf{T}_1 + 8 > \mathsf{T}_2$$
> $$\mathsf{T}_1 + 3 < \mathsf{T}_2, \quad .$$
>
> In order to do the same with a state transformation, one might invent a new function such as $\mathsf{Delay}(t_1, t_2, s)$ that maps a state into an identical state somewhere between t_1 and t_2 seconds later than the original. The above plan fragment could then be expressed as $\mathsf{B}(\mathsf{Delay}(3,8,\mathsf{A}(S_0)))$.
>
> Suppose one then decides that action B must also be more than 4 seconds after action A. In the residue approach, one simply adds another constraint to the design, namely $t_1+4 < t_2$. In the single-term approach, even though the function Delay has already been invented there is still a problem. Simply composing this additional constraint to give
>
> $$\mathsf{Delay}(4,8,\mathsf{B}(\mathsf{Delay}(3,8,\mathsf{A}(S_0))))$$

is not the correct state transformation. Instead, the system must somehow unpack the original composed function

$$\mathsf{B}(\mathsf{Delay}(3,8,\mathsf{A}(S_0)))$$

and know how to transform it to the desired term

$$\mathsf{B}(\mathsf{Delay}(4,8,\mathsf{A}(S_0))).$$

One might imagine such a program for taking a term (containing some fixed vocabulary) and an additional constraint and from these two elements outputting a new term incorporating that constraint, but to do so is ad hoc, at best. In fact, in order to make the above transformation, the system probably would have to unpack the term to something closely resembling a set of formulas.

In contrast, by expressing constraints on a design as a set of formulas, there is no need for rewriting the entire design every time that a new constraint is added. Instead of a problem to be solved in an ad hoc and probably awkward way, there is a simple way to add an additional design constraint, namely *set union*, and there already exists a well understood semantics for the language in which the design is expressed.

The same sort of problems arise in trying to express a partial ordering of actions, constraints on the values of components (restriction on the voltage drop across a given resistor, or restrictions on the allowed color or shape of a block, for example). Such constraints fit poorly into the single-term approach because they are not state transformations; they are facts about the design being constructed.

2.7.3 Reasoning about Partial Designs

Much of the rest of the thesis will concern itself with reasoning about a partially completed design during the design process. Residue procedures depend upon their ability to do consistency checking. Chapters 3 and 4 will require additional reasoning about a design.

Terms of a logic are indivisible objects; one can state facts about a whole term, but not about a part of a term. Thus, in order to reason about a design expressed as a single term, the composed functions expressing the design must be unpacked into a set of facts about which we can reason. In the residue approach this is obviously already the case — all information about the design are already expressed as a set of formulas; there is no need to transform the design into a form on which a system can reason.

2.7.4 Minimal Answers

It is not always the case that a completely specified design is needed. If order of two actions in a plan is immaterial, it is good to know this fact. By keeping options open, one allows choices to be made later when more information might be known, and one allows for more possibilities for implementing the assumables that are already specified. In the planning problem of Example 2.10, for example, if the order of the execution of the actions really did not matter, a parallel implementation would be possible. As a residue would not need to specify the ordering of the actions, the possibility of parallel execution would fall out of the residue approach.[15] As long as the parts of the design needed to prove sufficiency can be expressed as atomic formulas, there is no need for a residue to specify more.

2.7.5 Mimicking the Single-Term Approach with Residues

If it is desired to a residue procedure in a manner similar to the single-term approach, it is easy to do so. Suppose that instances of the relation EXECUTION are assumable. $\mathsf{EXECUTION}(a, s_i, s_f)$ means that action a will be executed in situation s_i transforming the world to sitation s_f. Writing axioms about state transformation operators in the form

$$\text{Prereqs}(a, s_i) \wedge \mathsf{Execution}(a, s_i, s_f) \supset \text{Postreqs}(a, s_f)$$

one gets the desired behavior.

Example 2.12 In Section 2.7.1 the following axiom was given for the PUTON using a single-term approach:

$$\forall a, b, s \;\; \mathsf{CLEAR}(a, s) \wedge \mathsf{CLEAR}(b, s) \supset \mathsf{ON}(a, b, \mathsf{PUTON}(a, b, s)).$$

To get the same behavior for a residue procedure, the above axiom would be written in the form:

$$\begin{array}{lll} \forall a, b, s_1, s_2 & \mathsf{CLEAR}(a, s_1) \wedge & \\ & \mathsf{CLEAR}(b, s_1) \wedge & \\ & \mathsf{EXECUTION}(\mathsf{PUTON}(a, b), s_1) \wedge & \\ & \mathsf{PRECEDES}(s_1, s2) \supset & \mathsf{ON}(a, b, s_2). \end{array}$$

[15]In this particular example, of course, the order of the execution really does matter.

2.7.6 Consistency Checking

Consistency checking is an integral part of the residue approach, but not the single-term approach. Why the difference?

At first glance, the answer seems to be that residues add something to the world model **W**, whereas single terms do not. In other words, the single-term approach is trying to prove that

$$\mathbf{W} \models \exists d\, \mathsf{G}(d),$$

where G is the design specification, whereas a the residue approach tries to add facts **D** to **W** such that

$$\mathbf{W} \cup \mathbf{D} \models \mathsf{G}. \tag{7}$$

However, one need only rewrite (7) as

$$\mathbf{W} \models (\mathsf{D} \supset \mathsf{G}),$$

where $\mathsf{D} = \bigwedge \mathsf{D}_i$ for all $\mathsf{D}_i \in \mathbf{D}$, to see that there must be some other difference. After all, the term d might specify an impossible design just as easily as the formulas **D**.

The real reason for the difference is the set of search paths considered by single-term approaches. As discussed in Section 2.7.2, single-term approaches can build up a design in one direction only. Due to the limitations of expression of a term, one cannot build up a design from the middle outward, but must either build up the design from the outputs to the inputs (as is commonly done) or perhaps from the inputs to the outputs.[16]

Let us assume that one is working backwards from the goal.[17] The original goal is completely *regressed* through each state transformation to give new goals[18]. Thus, at any time, the state transformations coming after (after in the design itself, not in the design process) have already been entirely specified (modulo variable bindings of arguments to the state tranformation operators) — it is impossible to choose which aspects of the design to consider in detail first. The importance of making design decisions in an acceptable order has been recognized since the beginning of AI. For example, one of the main ideas of GPS [71,23] was the notion of solving the goal spanning the "greatest difference" first. One may not do so under the restrictions of the single-term approach.

[16] In order to be able to design in other orders, one might consider a *hierarchical* approach as in Sacerdoti's ABSTRIPS [80]. Given such an approach, one can postpone decisions about the details of parts of the design. Unfortunately, just like residue methods, a hierarchical approach requires checking that the parts mesh as planned, i.e., consistency checking. It was a known limitation of ABSTRIPS that no such checking was done.

[17] The forward case is similar, and to the best of the author's knowledge, no such single-term system has been proposed.

[18] See Waldinger [97] or Nilsson [72] for an explanation of goal regression.

Example 2.13 If one has a goal of flying from New York to San Francisco, then taking a trip by ship to Hong Kong, and driving to a hotel in Hong Kong, it is hardly appropriate to plan the drive to the hotel, and then plan the boat trip, and finally plan the plane trip. One would almost certainly want to plan the cruise first, followed by the plane trip, followed by the drive. Instead, a unidirectional search through a set of state transformations forces one to plan the drive, then the cruise, and then the flight.

Single-term approaches generally assume that every composition of the allowed state transformation operators is a meaningful design. Because such terms are all that can be created, both during the design process and as the output of the design process, there is no need to check consistency. What is lost is the ability to consider state differences in any order, to specify partial orderings on operators, to insert application of operators between existing operators, or to partially specify operators[19].

If the above capabilities are not important, the world model for a residue system can be set up to mimic the approach of the single-term approach (as explained in Section 2.7.5), that is, one may perform a strictly backwards search just as in the single-term approach. Otherwise, one faces the need to design via successive refinements that may or may not be consistent.

2.8 Related Work

2.8.1 Reiter's Default Logic

Ray Reiter [74] develops what he calls a "Logic for Default Reasoning". His goal is to develop a logic for drawing plausible conclusions that are unprovable, but consistent with the initial world model. For example, if Fred is known to be a bird, Reiter's system will conclude that Fred can fly unless it can prove otherwise. Such a conclusion is made by using a *default*, which Reiter expresses as:

$$\delta_i = \frac{\alpha(\mathbf{x})\ :\ \mathbf{M}\beta_1(\mathbf{x}), \cdots, \mathbf{M}\beta_m(\mathbf{x})}{w(\mathbf{x})}. \tag{8}$$

The default δ_i is interpreted as, "If $\alpha(\mathbf{x})$ is true and if $\beta_1(\mathbf{x}), \cdots, \beta_n(\mathbf{x})$ can be consistently believed, then $w(\mathbf{x})$ may be believed.

[19]Such systems have also not generally dealt with specifications on the form of the design such as "No loops allowed" or "No more than four NAND-gates allowed." To handle such specifications, a single-term system would be forced to use some sort of ad hoc procedure to see if the design term meets this specification.

In the example above, the default was simply:

$$\frac{bird(\mathbf{x}) : \mathbf{M}\, flies(\mathbf{x})}{flies(\mathbf{x})}, \tag{9}$$

Reiter defines a *default theory* $\Delta = (D, W)$ to be a set of defaults D and a world model W. A default theory may have *extensions* E, that is, smallest sets of well-formed formulas such that:

1. $W \subseteq E$,

2. E is deductively closed, and

3. All defaults that *may* be added to E *are* in fact included in E.

In order to decide whether a formula p can be believed, Reiter asks, "Given p and $\Delta = (D, W)$, does there exists an extension E such that $p \in E$?" Unfortunately the problem is in general intractable. There is a subclass of the class of general default (that is, defaults expressed by (8)) such that there is a proof theory for the above question. This subclass is called the class of *normal defaults* and consists of defaults of the form:

$$\delta_i = \frac{\alpha(\mathbf{x}) : \mathbf{M}w(\mathbf{x})}{w(\mathbf{x})}. \tag{10}$$

Note that 9 is a normal default. For *normal default theories*, i.e., default theories such that all $\delta_i \in D$ are normal defaults, Reiter proves the following results:

1. All normal default theories have extensions.

2. For a formula p and a normal default theory Δ, there exists an extension E of Δ such that $p \in E$ *if and only if* there exists a top down default proof of p with respect to Δ.

Top down default proofs are Reiter's proof procedure for finding a subset $D_n \subseteq D$ such that for all formulas p, $p \in E \Leftrightarrow W \cup \mathsf{Consequents}(D_n) \vdash p$, where the Consequents of a default theory D is the set consisting of the $w(\mathbf{x})$ for each default $\delta_i \in D$.

Relation of Residue and Default Logic Both the residue approach and Default Logic begin with a world model W. In addition, there is a language of assumables $\mathcal{A}$ for residues corresponding exactly to Reiter's defaults D. In fact, each assumable can be expressed as:

$$\delta_i = \frac{: \mathbf{M}w(\mathbf{x})}{w(\mathbf{x})}. \tag{11}$$

The meaning of (11) is that $w(\mathbf{x})$ may be assumed if it is consistent to do so. It is seen from (10) and (11) that assumables are a proper subclass of the set of normal defaults, namely the class where the prerequisite $\alpha(\mathbf{x})$ is always true.

The fact that $\alpha(\mathbf{x}) = \mathsf{True}$ for all assumables means that Top Down Default Proofs reduce to a residue procedure. For this case, one need not construct a sequence of derivation trees, but only perform a single derivation, just as in a residue procedure. There is a small difference in that Reiter's proofs assume a control strategy called Linear Resolution (see Loveland [51] or Luchkam [53]) rather than set of support resolution. Linear Resolution seemingly was used only for the sake of simplicity. For set of support resolution, Reiter's proofs go through virtually unchanged. Although Linear Resolution is complete, the disadvantage of being forced to use Linear Resolution is that it can force the space to be searched in an inappropriate direction. For example, linear resolution can sometimes force forward chaining, which often is far branchier than backward chaining.

Whereas a residue consists of a set of assumables found necessary by the proof, Reiter defines *default support* as the set of defaults invoked in a given top down default proof. These are in exact correspondence.

The correspondence of designs to extensions is a bit trickier. If two different designs are inconsistent, then they cannot both belong to the same extension. On the other hand, there will in general be numerous extensions of which a given design is a member. A design only specifies part of the world; what happens outside of the design is irrelevant to the design, but changes the extension. For example, a plan might specify all the actions in the world from time $t = 0$ until $t = 10$, but says nothing about events after $t = 10$. Every inconsistent course of events after $t = 10$ will be in a separate extension.

Although residues are somewhat simpler than Reiter's proof theory, the approach unfortunately suffers from the same computational complexity as Reiter's work. Both are based on decision methods for first-order logic and thus are at best semi-decidable. However, neither is even semi-decidable because both depend upon proving satisfiability of the world model W unioned with the set D_n of assumptions made. As a result, both Default Logic (as Reiter points out) and residue procedures must rely on heuristic methods to become convinced of satisfiability.

2.8.2 Truth Maintenance

A major weaknesses of the residue procedures presented so far is that there is no way to learn from previous mistakes. The non-deterministic search procedure presented in Section 2.3

is neutral on the subject of caching results of previous deductions.[20] It does not specify a procedure for avoiding the pitfalls of blind search or of chronological backtracking, but does not tell us we cannot add such a procedure.

In his work on assumption-based truth maintenance systems (ATMS) [21,20,19] Johan de Kleer has listed a number of known problems with chronological backtracking. Let us review them here as they relate to Residue.

In *Futile Backtracking*, one finds a contradiction and backs up to a point that could already be eliminated as contradictory based on previously found contradictions. For example, if $\{x = 1, y = 1\}$ is a contradictory set, then backing up from a point $\{x = 1, y = 1, z = 0\}$ to another point $\{x = 1, y = 1, z = 1\}$ is futile. In a residue procedure, if some set G (consisting of assumables and remaining goals) causes a contradiction, i.e., $\mathbf{W} \cup \mathsf{G} \models$ `false`, then it can pay to note this "nogood" set (Steele [91]). Future set of assumables that either contain or entail the set G can be immediately eliminated.

Closely related is *rediscovering contradictions*. In *futile backtracking* we did not undo one of the choices in the newly discovered nogood set. In addition, we might find an old contradiction via a different path. As above, the caching of nogood sets will solve this problem.

Rediscovering inferences is a different problem. The fact that a deduction was performed on a different search path does not make it inapplicable on the present path. As long as the deduction was not based on a contradictory set of assumptions, then the deduction itself is valid whether of not the search path on which it was made was a dead end or not. In residue procedures, this is particularly important while checking consistency. There is no reason not to cache facts derived from **W** and various assumables in a global database to be used by all search paths (the subject of Chapters 3 and 4).

Incorrect ordering is de Kleer's name for finding contradictions at the right time. Of course, there is no general solution for this problem. In residue procedures, the problem expresses itself as the resource trade-off between goal reduction and consistency checking.

As de Kleer points out, a Truth Maintenance System (Doyle [22]) solves the first three problems. Since Doyle's TMS, there have been various other versions of TMS's, in particular McAllester [64,62,63], Martins [61], McDermott [67], and de Kleer [21,20]. Common to each of these systems is storing of the justications and assumptions upon which a deduction is based. However, they differ in the choice of information maintained. Doyle's "justification-based" TMS stores justification pointers to its immediate predessors and successors, but does not propagate new sets of support for each node at each step of the way. On the

2

other hand, de Kleer's "assumption-based" TMS (the ATMS) maintains a complete list of "environments", that is, assumption sets upon which each deduction might be based.[21] As a result, one can quickly see whether a node is a member of a given context (the deductive closure of $\mathbf{W} \cup G$ for some set of assumptions G. However, insertion of a new fact into such a database is expensive, as its effects on the various environments must propagate throughout the system. The justification-based TMS, on the other hand, maintains a single consistent context at all times. As long as contradictions are not found, insertion of new facts is quite inexpensive. However, when contradictions are found, the TMS may need to do a good deal of search in order to find a new consistent context.

2.8.3 Douglas Smith

In [87] Douglas Smith used a natural deduction system for program synthesis, using a modified single-term approach in extracting the answer from the proof tree. Rather than prove that the generated program was true for all preconditions, Smith attempted to reduce his goal to a set of preconditions that fit into one of a number of existing skeletons. The skeletons were not assumable, but rather were attached to procedures for modifying both the remaining preconditions and for extracting the program from the existing derivation. See [88,89].

2.8.4 PROLOG/EX1

In PROLOG/EX1 [101,100], Adrian Walker modified Prolog such that in case of failure to produce an answer, the system returns an explanation of what additional facts would be needed to produce an answer. The elements of such an explanation of failure are analogous to assumables in residue procedures. As such, Walker faced the problem of deciding what sufficient set facts are most reasonable to assume to explain the failure. Not just any set of facts will do — the query itself obviously suffices for producing a proof of the query, but is hardly an acceptable explanation of why the proof failed. Walker's approach was to use a set of three domain-independent rules for deciding what proof steps to assume based upon depth in the proof tree and upon constants in the query.

2.8.5 Theorist

At the University of Waterloo, David Poole, Randy Goebel and associates have recently developed a system called Theorist [29,73] for theory formation problems. As Poole et al

[21] Possible only because de Kleer deals only with propositional calculus.

have pointed out, Theorist's theories are similar to residues, and they also have pointed out the similarity with Reiter's Default Theories [74]. There is also a good deal of similarity between assumability and that of appropriateness of an explanation for diagnosis and/or theory formation problems. Just as the constitution of a legal design is rather arbitrary, the same holds true for an explanation of a problem. At some level, either the system or the system designer must legislate what consititutes a sufficient explanation that need not be further explained, just as we legislate what design assumptions can be made without further explanation. Luckily, for design, the problem of deciding what are the primitive components of a design seems much easier than deciding what constitutes a primitively acceptable explanation. See Charniak and McDermott [16] and especially McDermott [68] for pessimistic views on finding such criteria for explanation.

2.9 Conclusion

This chapter presents the residue approach to design synthesis and two procedures for finding residues. By using a set of atomic formulas to express designs rather than using a single term, one gains two important advantages: (1) Residues can easily express much finer-grained decisions than systems based upon a single term, and (2) Parts of designs can be specified in any order rather than strictly from the goal backwards or from the initial state forward. In addition, the representation of partially complete designs as sets of facts enables an inference system using predicate calculus notation to reason directly *about* the design. For a given system if the full generality of the residue appraoch is not needed, the database can be written in such a way that the residue approach reduces to a system isomorphic to the single-term approach.

Residue procedures have been used in a number of projects at Stanford — DART [28] used residues to generate diagnostic tests for combinatoric circuits and in his PhD thesis, Narinder Singh [83] used a residue procedure for generating diagnostic tests for IBM Printer Adapter cards. Residues were also used in Russ Greiner's analogy understanding program [34] and in Jock Mackinlay's APT [55,56] tool for automatic graphical presentation.

The generality of the residue approach contains the seeds of a potential combinatoric explosion. In particular, in the most general case residue procedures require a consistency check, which is a non-semidecidable problem. Usually, however, careful crafting of the knowledge base prevents consistency checking from being prohibitively expensive.

Chapters 3 and 4 show how the full generality of the residue approach can sometimes be exploited to bring about large reductions in the search space.

Chapter 3

Supersumption

3.1 Ramifications of a Goal

It is well-nigh impossible to drive a car without consuming fuel. Similarly, writing a new version of a disk file will change numerous parameters associated with that file such as write-date, disk address, and size. In both cases, the latter condition inevitably accompanies the former; the latter is a *ramification* of the former.

Making the assumption that the world's behavior (or at least the relevant portion of it) can be modelled by a set of first-order axioms $\mathbf{W}$, ramifications, such as the above, can be captured by logical implication. In other words, if $\mathbf{W}$ is to capture fuel consumption as a ramification of driving, a formula denoting consumption of fuel must be a logical implication of any formula denoting driving of a car.

A goal G is a partial description of a state. If some formula N[1] is logically implied by $\mathbf{W} \cup \mathsf{G}$, then in every state $\mathcal{S}$ described by G, whatever is denoted by N also holds; N is a ramification of G's being true. The process of finding ramifications of a goal can be seen as (1) looking at a partial description of a hypothetical state $\mathcal{S}$ in which G is holds and (2) trying to fill in more of the description of $\mathcal{S}$ based upon our knowledge of consistent or "allowed" states of the world (where this knowledge is captured in $\mathbf{W}$).

Obviously, it may take an arbitrary amount of inference to find a particular ramification N. Some facts follow immediately from G while others may require a complicated line of reasoning to discover. Of course, the "distance" between a goal G and ramification N is a function of the database and of the inference engine at hand. Having a large file in one's directory almost immediately leads one to realize that it will be expensive to keep the file. On the other hand, it may take a good deal more inference to see that the presence of that

[1]The symbol N is being used for a ramification instead of R to avoid mistaking the symbol for a residue.

file precludes receiving any additional electronic mail.[2]

3.2 Using Ramifications of a Goal

There are a number of ways in which to use the known ramifications of a given goal's being true. First and foremost is *elimination of an inconsistent goal* — if a ramification of a goal's being true is known to be impossible, then the goal itself is not achievable. Reduction of problems to the Halting Problem is a good example of this. Given some goal G, if we show that achieving G implies that we can solve the Halting Problem, then G is impossible to achieve. A more mundane example would be to try to find a disk file that is larger than 1 MByte. If it is known that all files are stored on 256 KByte floppies, it can immediately be said that there can be no such file. On the other hand, if the ramifications of the goal being true are ignored, all the files in the system will probably be enumerated, checking the size of each to see if it is larger than 1 MByte. Of course, no such file will be found.

Another use of ramifications is *restriction of the search space via additional constraints.* Suppose again that a file that is larger than 1 MByte is sought. This time, however, there is a 50 MByte hard disk and hundreds of 256K floppy disks. From the above information, it can be derived that a file larger than 1 MByte must reside on the hard disk. Thus, files that reside on the floppies need not be considered, and a good deal of search has been avoided. Elimination of an inconsistent goal is a special case of restriction via additional constraints, that is, the case in which the entire search space is eliminated from consideration rather than just part of it.

A third use of ramifications is in *enabling additional heuristics to be used.* It may be the case that whatever heuristics used are not directly applicable to the goal at hand, but *are* directly applicable to some ramification of the goal. Without filling out more of the description of the goal state, the use of potentially applicable heuristics may be missed. Suppose we are looking for an executable file for playing chess. Any solutions for the goal will also fall into the category of being executable files for playing a game. If it is known that *most* executable game files are located on directory **/usr/games**, then we would be well-advised to look here first for such a file. Note that without realizing that chess is a game (that is, finding a ramification of the goal), we would not have known that the heuristic information applies (that is, that most executable game files reside on /usr/games).

[2]This situation often arises in operating systems such as TOPS-20 that impose a hard limit on the amount of disk space a user is allowed to have.

3.3 Subgoals, Design Decisions and Ramifications

The discussion so far has been about ramifications of a given goal being true. The discussion applies, however, not just to the top level goal of any problem, but to any subgoal[3] generated in trying to solve the problem.

> **Example 3.1** Consider the goal G of being in Denver, starting from a state of being in San Francisco. One way to achieve G is via a commercial airline flight from San Francisco to Denver, that is, a subgoal S to take a flight from San Francisco to Denver. If the cheapest such flight costs \$150, then a ramification N of S being true is that at least \$150 will be spent. On the other hand, N is not a ramification of G being true; there might well be a way to get from San Francisco to Denver for less money, say \$100.

The above example shows that *ramifications of achieving a subgoal are not necessarily ramifications of achieving the original goal.* Achievement of a subgoal may have additional ramifications because the subgoal may be more restrictive than the original goal — there is additional information from which to reason. In Example 3.1, in reducing G to S there was a *desigr decision* made, namely, to take a commercial flight. Since S's ramification[4] that the trip will cost at least \$150 was based upon this design decision, the ramification will not necessarily hold for the original goal G.

The process of goal reduction can be viewed as a process of making design decisions restricting the class of solutions to consider. Although most of the previous examples have been about ramifications of a top-level goal, anything that can be said about using ramifications of a top-level goal can also be said about using ramifications of a subgoal. All ramifications of a subgoal will not necessarily hold for the top-level goal, but since the

[3] There is potential confusion in that the logic programming community uses *subgoal* to refer to a conjunct of a goal (which is a conjunction of literals). Here, *subgoal* is used to mean an entire goal (usually a conjunction) to which another entire goal has been reduced.

[4] The word "ramification" usually refers to one thing being true as part and parcel of another's being true. One might speak of the "ramifications of a fact being true," i.e, "ramifications of *achieving* a goal," and one might also speak of the "ramifications of *having* a given goal." In the interest of brevity let us make the convention that "ramification of a goal G" refers to a ramification of *achieving* the goal G, rather than a ramification of *having* the goal G. Such usage is consistent with viewing a goal G as a partial description of a desired state. It is also natural to speak of "ramifications of a design decision," and again, what is meant is a ramification of *implementing* the design decision rather than a ramification of *making* the design decision. Since both goals and design decisions are represented as formulas (See Chapter 2), there is no formal need to distinguish between ramifications of goals and ramifications of design decisions — it is only important that it be understood that ramifications of a goal or design decisions refer to *achieving* a goal and *implementing* the design decisions.

subgoal is a *sufficient condition* for the top-level goal, any ramification of the top-level goal will be a ramification of any of its subgoals. Later it will be shown (Section 4.7) that there are cases in which a subgoal is derived from more than one higher-level goal and that a higher-level goal may have ramifications that are not ramifications of the subgoal.

3.4 Formal Definition of Ramifications

The previous sections have discussed in broad terms how ramifications of a goal being true can be useful in pruning the search space of a problem. Let us now give a precise formulation of a ramification a goal's being true. Although it is tempting to simply say that a ramification is any formula N for which $\mathbf{W} \models (\mathsf{G} \supset \mathsf{N})$, the existence of variables requires a more careful definition.

Definition 3.1 (Ramification) *Let* $\mathbf{W}$ *be a satisfiable set of closed formulas, and let* G *be a closed formula in prenex normal-form, that is,* $\mathsf{G} = \Box_1 x_1 \ldots \Box_m x_m \mathbf{G}$, *where* $\Box_i$ *is either* $\exists$ *or* $\forall$, *and* $\mathbf{G}$ *is a quanitifier-free formula whose only free variables are* $x_1, \ldots, x_g$. *Suppose* N *is a formula whose only free variables are* $y_1, \ldots, y_n$ *such that* $\{y_1, \ldots, y_n\} \subseteq \{x_1, \ldots, x_g\}$. *We say that* N is a ramification of G given $\mathbf{W}$ *(or* Ramification$(\mathbf{W}, \mathsf{G}, \mathsf{N})$*) if*

$$\mathbf{W} \models \forall x_1 \ldots \forall x_g (\mathbf{G} \supset \mathsf{N}). \tag{12}$$

In addition, we say that N is a strong ramification of G given $\mathbf{W}$ *(or* StrongRam$(\mathbf{W}, \mathsf{G}, \mathsf{N})$*) if* Ramification$(\mathbf{W}, \mathsf{G}, \mathsf{N})$ *and* $\mathbf{W} \not\models \forall y_1 \ldots \forall y_n \mathsf{N}$..

Note that for every formula N containing free variables $z_1, \ldots, z_q$ not in $\mathbf{G}$ there is obviously a corresponding formula $\mathsf{N}' = \forall z_1 \ldots z_q \mathsf{N}$. As a result, the restriction that N's free variables be a subset of G's free variables causes no loss of generality.[5]

Example 3.2 Consider a goal $\mathsf{G} = \exists x \exists y\ \mathsf{A}(x) \wedge \mathsf{B}(x, y) \wedge \mathsf{C}(y)$ and $\mathbf{W}$ containing a formula

$$\forall u \forall v \forall w\ (\mathsf{A}(u) \wedge \mathsf{C}(v)) \supset \mathsf{D}(u, v, w).$$

Then the open formula $\forall w \mathsf{D}(x, y, w)$ is a ramification of G given $\mathbf{W}$.

[5] In Chapter 4, where ramifications will be viewed as clauses, it will be necessary that unquantified variables appear in ramifications that do not appear in the goal formula. This is only an artifact of representation via clauses; these variables are implicitly universally quantified.

Example 3.3 Consider the goal

$$\mathsf{G} = \exists x \mathsf{Flies}(x) \wedge \mathsf{Hairy}(x) \wedge \mathsf{Loves}(x, \mathsf{Bertha}). \tag{13}$$

Suppose the database **W** contains the formula

$$\forall\, y\ [\mathsf{Flies}(y) \wedge \mathsf{Hairy}(y) \supset \mathsf{Bat}(y)]. \tag{14}$$

Since follows from (14) that

$$\mathbf{W} \models \forall\, x\ [\mathsf{Flies}(x) \wedge \mathsf{Hairy}(x) \wedge \mathsf{Loves}(x, \mathsf{Bertha}) \supset \mathsf{Bat}(x)], \tag{15}$$

$\mathsf{Bat}(x)$ is a ramification of G given **W**.

The literal $\mathsf{Loves}(x, \mathsf{Bertha})$ is not needed to show that $\mathsf{Bat}(x)$ is a ramification of G and has no effect on whether $\mathsf{Bat}(x)$ is a ramification of G. Similarly we might be able to derive other ramifications dependent upon $\mathsf{Loves}(x, \mathsf{Bertha})$, but with no dependence upon $\mathsf{Hairy}(x)$.

In the previous examples the ramification was provable via application a single application of modus ponens using a proposition from **W**. This restriction need not hold, as in the following example:

Example 3.4 Consider a goal formula

$$\mathsf{G} = \mathsf{Travel}(\mathsf{Palo\ Alto}, \mathsf{Denver}) \wedge \mathsf{Duration}(\mathsf{Palo\ Alto}, \mathsf{Denver}) \leq 4, \tag{16}$$

that is, the task at hand is to plan a trip from Palo Alto to Denver that takes 4 hours or less. Suppose **W** contains axioms (17) - (20):

$$\begin{array}{rll} \forall\, x, y, d, t & & \mathsf{Travel}(x, y) \\ & \wedge & \mathsf{Dist}(x, y) \geq d \\ & \wedge & \mathsf{Duration}(x, y) \leq t \\ & \supset & \mathsf{AvgSpeed}(x, y) \geq \frac{d}{t}, \end{array} \tag{17}$$

that is, the average speed required for a trip is greater than or equal to the distance covered divided by the maximum time allowed for the trip,

$$\begin{array}{rll} \forall\, x, y, s_1, s_2, m & & \mathsf{AvgSpeed}(x, y) = s_1 \\ & \wedge & \mathsf{MaxSpeed}(m) = s_2 \\ & \wedge & s_1 > s_2 \\ & \supset & \mathsf{Mode}(x, y) \neq m, \end{array} \tag{18}$$

that is, if the average speed required in getting from x to y is greater than the maximum speed for a given mode of travel, then that mode of travel will *not* be used for the this portion of the trip.

$$\mathsf{Dist(Palo\ Alto, Denver)} = 1000, \tag{19}$$

that is, the distance between Palo Alto and Denver is 1000 miles, and

$$\mathsf{MaxSpeed(Auto)} = 70, \tag{20}$$

that is, the maximum speed for an automobile trip is 70 miles per hour.

Consider the formula $\mathsf{N} = \mathsf{AvgSpeed(Palo\ Alto, Denver)} = 250$. Since $\mathbf{W} \models (\mathsf{G} \supset \mathsf{N})$, that is,

$$\begin{aligned} \mathbf{W} \models\ & [\mathsf{Travel(Palo\ Alto, Denver)} \wedge \mathsf{Duration(Palo\ Alto, Denver)} \leq 4] \\ & \supset \mathsf{AvgSpeed(Palo\ Alto, Denver)} = 250, \end{aligned}$$

N is a ramification of G. Similarly,

$$\mathsf{Mode(Palo\ Alto, Denver)} \neq \mathsf{Auto} \tag{21}$$

is also a ramification of G.

Ramifications in the Residue Approach Chapter 2 discussed design as a problem of finding a residue, that is, a problem of reducing a goal G to a subgoal $\mathsf{D}_1 \wedge \ldots \wedge \mathsf{D}_d$ such that for $\mathbf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_d\}$

1. $\mathbf{W} \cup \mathbf{D} \models \mathsf{G}$
2. Each D_i is assumable.
3. $\mathbf{W} \cup \mathbf{D}$ is satisfiable.

Goals and designs are distinguished only by the property of *assumability* (See Section 2.2.3). A legal design is a goal that happens to be assumable. Because residue procedures express design decisions as part of a goal, there is no need to distinguish between ramifications of a goal being achieved and ramifications of making a design decision — both can be derived from $\mathbf{W}$ and from a given goal (or subgoal) generated by the residue procedure.

3.5 Supersumption

The key idea of this chapter is *supersumption*,[6] the reformulation of a goal by appending of additional constraints. In this thesis, all such added constraint will be ramifications, but it is reasonable to consider adding other constraints to a goal, for example, constraints that *probably* follow from the goal.

Definition 3.2 (Supersumption) *Given closed well-formed formulas* $\mathsf{G} = \Box x_1 \ldots, \Box x_g \widehat{\mathsf{G}}$ *and* G', *where* $\widehat{\mathsf{G}}$ *is quantifier-free,* G' supersumes G *(or "*G' *is a supersumption of* G*") if*

$$\mathsf{G}' \equiv \Box x_1 \ldots \Box x_g(\widehat{\mathsf{G}} \wedge \mathsf{A}),$$

where A*'s only free variables are* $\{y_1, \ldots, y_a\}$ *and* $\{y_1, \ldots, y_a\} \subseteq \{x_1, \ldots, x_g\}$.

Definition 3.2 only requires logical equivalence between $\Box x_1 \ldots \Box x_g(\widehat{\mathsf{G}} \wedge \mathsf{A})$ and G'. It does not require that G' actually be the formula $\mathsf{G} \wedge \mathsf{A}$. In particular, if $\widehat{\mathsf{G}}$ and A are conjunctions, this definition says nothing about the ordering of the conjuncts in G'.

If A is a ramification of G, then the same set of bindings for $\mathbf{x}$ cause G and G' to be entailed by $\mathbf{W}$.

Theorem 3.1 *For a goal* $\mathsf{G} = \Box x_1 \ldots, \Box x_g \widehat{\mathsf{G}}$, *world model* $\mathbf{W}$ *and* Ramification($\mathbf{W}$, G, N), *every model of* $\mathbf{W}$ *and* $\widehat{\mathsf{G}}$ *is a model of* $\widehat{\mathsf{G}} \wedge \mathsf{N}$.

Proof: Because N is a ramification of G, we know that

$$\mathbf{W} \models \forall x_1 \ldots \forall x_g(\widehat{\mathsf{G}} \wedge \mathsf{N}). \tag{22}$$

Let $\mathcal{M}$ be an arbitrary model of $\mathbf{W}$ and $\widehat{\mathsf{G}}$. By the deduction theorem and (22), $\mathcal{M}$ must also be a model for N, and therefore for $\widehat{\mathsf{G}} \wedge \mathsf{N}$. Thus,

$$\mathbf{W} \models \Box x_1 \ldots \Box x_g(\widehat{\mathsf{G}} \wedge \mathsf{N}).$$

∎

note that Theorem 3.1 did not preclude $\mathbf{W}$ containing a set $\mathbf{D}$ of formula such that $\mathbf{W}' \cup \mathbf{D} \models \mathsf{G}$, where $\mathbf{W}' = \mathbf{W} - \mathbf{D}$. In other words, we have the following corollary:

[6]The motivation for the term *supersumption* is that it is in some sense opposite to *subsumption* in which a disjunction C (the denial of a goal) is matched with a known formula consisting of a subset of C's disjuncts. Supersumption, on the other hand, *adds* conjuncts to a goal; it creates a goal G′ the denial of which is subsumed by the denial of the original goal G. If one considers supersumptions of non-conjunctive goals then the analogy breaks down.

Corollary 3.2 *If* **D** *is a residue of* $\mathsf{G} = \Box x_1 \ldots \Box x_g \widehat{\mathsf{G}}$ *given world model* **W**, *and if* Ramification(**W**, G, N), *then* **D** *is also a residue of of* $\Box x_1 \ldots \Box x_g(\widehat{\mathsf{G}} \wedge \mathsf{N})$.

Example 3.5 Consider the goal of trying to find an executable file named chess on some computer system:

$$\mathsf{G} = \exists x \mathsf{File}(x) \wedge \mathsf{Name}(x, \mathsf{chess}) \wedge \mathsf{Executable}(x). \tag{23}$$

One possible supersumption of G would be formed by adding an additional constraint A = Directory-of(x, /usr/games) to form a new goal

$$\begin{aligned} \mathsf{G}' = \exists x \ & \mathsf{File}(x) \wedge \\ & \mathsf{Name}(x, \mathsf{chess}) \wedge \\ & \mathsf{Executable}(x) \wedge \\ & \mathsf{Directory\text{-}of}(x, /\mathsf{usr}/\mathsf{games}). \end{aligned} \tag{24}$$

If A = Directory-of(x, /usr/games) is a ramification of the goal, then the new goal G′, created by adding the conjunct Directory-of(x, /usr/games) to G, has the same set of solutions as G.

3.6 Speedup Via Supersumption

As discussed in Section 3.2, supersumption can reduce a search space by (1) eliminating inconsistent goals, (2) restricting search space via additional constraints, and (3) allowing additional heuristics to be used. The first and third of these mechanisms are easily understood. In this section, the second mechanism, restriction of the search space, is discussed in more detail.

The purpose of supersumption is to reformulate a goal G as a new goal G′ that is cheaper to solve than G. As in all reformulations, there is a saving if

$$\mathsf{Cost}(\mathsf{Reformulation}(\mathsf{G})) + \mathsf{Cost}(\mathsf{Solving}(\mathsf{G}')) < \mathsf{Cost}(\mathsf{Solving}(\mathsf{G})). \tag{25}$$

In other words, the savings in finding solutions for G′ must more that offset the overhead of reformulating G. For our purposes, "Solving(G)" can refer to finding solutions by either abduction or deduction, and can refer to problems of finding one solution or finding all solutions (assuming a finite number of solutions).

Unfortunately, estimating the cost of solving a given problem is not a well developed area. In order to make the problem at all tractable, let us make the following assumptions:

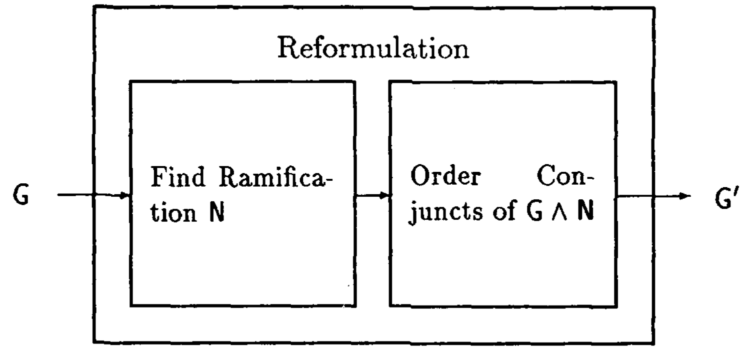


Figure 7: Reformulation of Conjunctive Goals via Supersumption

1. G and G′ are ordered conjunctive goals. A *conjunctive goal* is a closed formula $g = \Box x_1 \ldots \Box x_g(g_1 \wedge \ldots \wedge g_n)$, where each g_i is an atomic formula. Thus $\exists x \forall y \mathsf{A}(x) \wedge \mathsf{B}(x, y)$ is a conjunctive goal, whereas $\exists x \forall y(\mathsf{A}(x) \vee \mathsf{B}(x, y)) \supset \mathsf{D}(x, y)$ is not. Just as clauses in resolution can be seen as being either sets or sequences of literals, so can conjunctions. An *ordered* conjunctive goal is a *sequence* of atomic formulas. In an ordered conjunctive goal it is meaningful to refer to the first (leftmost) conjunct and to speak of stepping though the conjuncts one at a time from the first to the last (or rightmost) conjunct.

2. Conjuncts will be solved in order starting at the left, that is, the leftmost conjunct is solved independent of the remaining conjuncts, and any constraints needed for the solution to the leftmost conjunct (variable bindings or assumptions made) are imposed upon the remaining conjuncts, and the process begun anew.

For problems for which the above two assumptions hold, the supersumption process is illustrated by Figure 7. For such goals G, one first creates a new goal $\mathsf{G} \wedge \mathsf{N}$ and then orders the conjuncts, yielding an alternative goal G′.

Chapter 4 deals with finding ramifications for a given conjunctive goal G and world model **W**. The subject of ordering conjunctive goals has received attention in both the database and AI literatures. The approaches that have generally made the following assumptions:

3. All solutions $\{y_1 \rightarrow Y_1, \ldots, y_j \rightarrow Y_j\}$ to $\mathbf{W} \models \mathsf{G}$ are sought, where $\{y_1 \ldots, y_j\}$ is a subset the set of the existentially quantified variables in G and the Y_i are ground

instances of the y_i.[7]

4. For every conjunct $\mathsf{R}(T_1, \ldots, T_n)$ appearing in either G or in G′, R is either *evaluable* or *extensional*, where R is a relation symbol and T_i is a term. An *evaluable* relation is such that the truth value of any ground instance of an atomic formula containing this relation can be ascertained in constant time. Examples of evaluable relations on integers are *less-than*, *oddp*, or *positivep*. *Extensional* relations are such that all known ground instances of each conjunct containing this relation appear directly in **W**; no additional ground instances of the conjunct are entailed by **W**. In short, each conjunct appearing in G or G′ may simply be looked up in **W** to find all of its known ground instances.

As stated above, the subject of conjunct ordering has been studied in under most or all of the above assumptions. Authors such as Blasgen and Eswaren [3], King [44] and Chakravarthy [13] make Assumptions 1-4 in their work, as is common in the database literature. David E. Smith [86,85] adds another assumption, that every extensional relation is indexed on each of its arguments.

This research assumes that there exists the means to order conjuncts reasonably. Assumptions 1 and 2 are made, but Assumptions 3-5 will be explicitly stated if they are being assumed.

3.6.1 Generators and Filters

It will be useful to distinguish two ways that a conjunct can act with respect to a variable, as a *generator* or as a *filter* of its values. If the ordered conjunction is to be solved in order, then for every variable x in every conjunct C, it can be said whether x will be grounded or not when solutions are to be found for C. The first (leftmost) appearance of each variable will not be grounded and the rest grounded. Thus, the first conjunct C in which variable x appears *generates* values for x; we say that C is a *generator*[8] for x. For any subsequent conjuncts D in which x appears there will be solutions for some subset of the values of x

[7] Note that this assumption disallows indefinite, i.e., disjunctive solutions. See Reiter [75] for a discussion of indefinite solutions.

Note also that this assumption is not very strong in that the cost of finding one solution can usually be reasonably approximated as the time of finding all solutions divided by the number of solutions.

[8] In programming, given a problem (or subproblem) with many solutions $S_1, S_2, \ldots$, a *generator* is the name commonly given to a procedure that returns one solution each time it is called, and if the S_i are exhausted, returns a token saying that it can produce no more solutions. Generators are commonly implemented via coroutines, Algol *own* variables, reference to global data structures, or some other form of memory between calls.

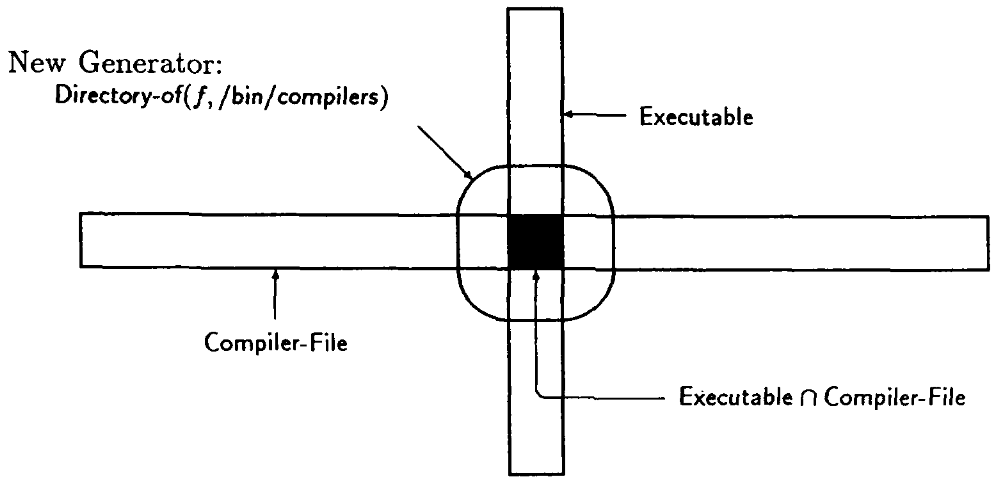


Figure 8: Speedup Obtained Using Additional Constraint as Generator

generated by C; we say that D is a *filter* for x. Note that a conjunct can be a generator for some variables and a filter for others.

Example 3.6 Given the ordered conjunctive goal $\mathsf{C}(x) \wedge \mathsf{D}(x,y) \wedge \mathsf{E}(x,y)$, $\mathsf{C}(x)$ acts as a generator for x, $\mathsf{D}(x,y)$ acts as a filter for x and a generator for y, and $\mathsf{E}(x,y)$ acts as a filter for both x and y.

3.6.2 Ramifications as Generators

One of the ways in which a supersumption $\mathsf{G}' \equiv \mathsf{G} \wedge \mathsf{A}$ can be cheaper to solve than G itself is for the additional constraint A to act as a generator of values for some variable x.

Let us consider a very simple case in which the cost of finding all solutions to a conjunct C is equal to the number of ground instances of C appearing in **W**. The above assumption corresponds roughly to a database lookup on a fully indexed extensional relation.

Example 3.7 Consider a database **W** of information about the files of a computer system. Suppose that all solutions to each of the atomic formulas $\mathsf{Executable}(f)$,

Compiler-File(f), and Directory-of(f, d) can be found with cost equal to the number of solutions. Here, Executable is a unary relation holding for all executable files, Compiler-File is a unary relation that holds for all files of source code, binary code or documentation for compilers of all programming languages, and Directory-of is a binary relation such that the second argument is the name of the directory on which the file named by the first argument is found. Only if d is a ground term is it assumed that the cost of generating all solutions to a query Directory-of(f, d) is equal to the number of solutions.

Given the above, suppose we have the query

$$\mathsf{G} = \exists f \mathsf{Executable}(f) \wedge \mathsf{Compiler\text{-}File}(f),$$

that is, a request has been made to list all executable compiler files. Assuming the number of executable files is less than the number of compiler files, the goal G is already optimally ordered. G uses Executable(f) as the generator of values for f and filters this set of values with Compiler-File(f). The cost of finding all solutions to G would be

$$\begin{aligned}\mathsf{Cost}(\mathsf{Solving}(\mathsf{G})) &= \mathsf{Card}(\mathsf{Executable}(f)) + \\ &\quad \mathsf{Card}(\mathsf{Executable}(f) \cap \mathsf{Compiler\text{-}File}(f)),\end{aligned}$$

and

$$\mathsf{Card}(\mathsf{Executable}(f)) \leq \mathsf{Cost}(\mathsf{Solving}(\mathsf{G})) \leq 2 * \mathsf{Card}(\mathsf{Executable}(f)),$$

where Card(s) is the cardinality of the set s.

Now, suppose that it is known that all executable compiler files are on directory /bin/compilers. For example, **W** might contain a formula

$$\forall y\ \mathsf{Executable}(y) \wedge \mathsf{Compiler\text{-}File}(y) \supset \mathsf{Directory\text{-}of}(y, /\mathsf{bin}/\mathsf{compilers}).$$

In such a case, N = Directory-of(f, /bin/compilers) would be a ramification of G, yielding one possible supersumption

$$\mathsf{G}' = \exists f \mathsf{Directory\text{-}of}(f, /\mathsf{bin}/\mathsf{compilers}) \wedge \mathsf{Executable}(f) \wedge \mathsf{Compiler\text{-}File}(f).$$

The cost of finding all solutions to G′ is

$$\begin{aligned}&\mathsf{Cost}(\mathsf{Solving}(\mathsf{G}')) = \\ &\quad \mathsf{Card}(\mathsf{Directory\text{-}of}(f, /\mathsf{bin}/\mathsf{compilers})) + \\ &\quad \mathsf{Card}(\mathsf{Directory\text{-}of}(f, /\mathsf{bin}/\mathsf{compilers}) \cap \mathsf{Executable}(f)) + \\ &\quad \mathsf{Card}(\mathsf{Directory\text{-}of}(f, /\mathsf{bin}/\mathsf{compilers}) \cap \mathsf{Executable}(f) \cap \mathsf{Compiler\text{-}File}(f)).\end{aligned}$$

Assuming that

$$\mathsf{Card}(\mathsf{Directory\text{-}of}(x, \mathsf{/bin/compilers})) \ll \mathsf{Card}(\mathsf{Executable}(x))$$

it can be seen that

$$\begin{aligned} \mathsf{Cost}(\mathsf{Solving}(\mathsf{G}')) &\leq \mathsf{Card}(\mathsf{Directory\text{-}of}(f, \mathsf{/bin/compilers})) + \\ &\quad 2 * \mathsf{Card}(\mathsf{Executable}(f)) \\ &\ll \mathsf{Cost}(\mathsf{Solving}(\mathsf{G})). \end{aligned}$$

The above example is illustrated in Figure 8.

In the above example, it is assumed that the cost of solving a conjunct is equal to the number of solutions. Such an assumption essentially means that there is no search for solutions — they are simply enumerated. In situations where more search is involved, supersumption can be a more powerful tool.

Example 3.8 Consider the goal

$$\mathsf{G} = \exists x \mathsf{Filename}(x, \mathsf{Chess}) \wedge \mathsf{Executable}(x),$$

that is, find the set of all executable files named chess. This time, let us assume that both $\mathsf{Filename}(x, \mathsf{Chess})$ and $\mathsf{Executable}(x)$ can be solved with cost proportional to the number of files in the entire computer system, typically $\mathcal{O}(10^5)$. Ground instances of both of these conjuncts can be solved with unit cost. The above cost assumptions are a reasonable model of most computer systems. A list of all files with a given name is not usually directly available on most systems. To get such a list one must somehow enumerate all the files on the system and then filter this list to find those with a given name. Thus, assuming that one knows how to enumerate all the files of the entire system (without further search for such a procedure), one can find the set of files named Chess in time proportional to the total number of files.

Given such a goal, a human user would typically either find a short cut based on other available information, or else resign himself to a large search. Let us see how supersumption can use such additional information in order to reduce the search.

Suppose now that it is known that

$$\mathsf{Game(Chess)}$$

and that

$$\forall y\, \mathsf{Game}(y) \wedge \mathsf{Executable}(y) \supset \mathsf{Directory\text{-}of}(y, \mathsf{/usr/games}),$$

that is, all executable files for games are located on directory /usr/games. In such a case, $\mathsf{N} = \mathsf{Directory\text{-}of}(x, \mathsf{/usr/games})$ is a ramification of G, and the goal

$$\mathsf{G}' = \exists x \mathsf{Directory\text{-}of}(x, \mathsf{/usr/games}) \wedge \mathsf{Filename}(x, \mathsf{Chess}) \wedge \mathsf{Executable}(x)$$

is the obvious reordering of $\exists x(\widehat{\mathsf{G}} \wedge \mathsf{N})$, where $\widehat{\mathsf{G}} = \mathsf{Filename}(x, \mathsf{Chess}) \wedge \mathsf{Executable}(x)$.
In constrast to the other two conjuncts, $\mathsf{Directory\text{-}of}(x, \mathsf{/usr/games})$ asks for information that is directly available to a typical system, that is, the solutions to it can be enumerated with cost proportional to the number of solutions to the conjunct. Typically, a directory will contain $\mathcal{O}(10^2)$ files, and thus

$$\mathsf{Cost(Solving(G'))} \approx 10^2 + 1 + 1 \approx 10^2.$$

A speedup of three orders of magnitude seems quite large, but for such a problem it is not unrealistic — an unreasonable goal was given to the system. Using available information, one is able to reformulate such a goal to yield much more reasonable one. If all goals given to a system were formulated in the optimal fashion, there would be no need for many sorts of optimizations. Indeed systems would have to be far less robust. But, *it is for precisely such badly stated goals that techniques such as supersumption are needed — to allow a problem-solving system to handle a wider range of goals with acceptable speed.*
The above examples contained only a single variable. In addition, the additional constraint was added in front of the first conjunct of the original query G. In general, however. there may be many variables, and the additional constraint A need not appear as the first conjunction in order to act as a generator for some variable x_j.

3.6.3 Additional Restrictions on Arguments

A related way in which ramifications can help in generation is by **restriction of additional variables** on a given database lookup.

Example 3.9 Suppose **W** contains ground formulas of the form

$$\mathsf{Teaches}(\langle\mathsf{Instructor}\rangle, \langle\mathsf{Subject}\rangle, \langle\mathsf{Hour}\rangle, \langle\mathsf{room}\rangle), \tag{26}$$

and the following query is posed:

$$\mathsf{G} = \exists i \exists r \mathsf{Teaches}(i, \mathsf{Math}, \mathsf{1pm}, r). \tag{27}$$

If it is known (or can be derived) that Fred is the only teacher of afternoon math courses, i.e.,

$$\forall\, i, h\ \mathsf{Teaches}(i, \mathsf{Math}, h, r) \wedge \mathsf{Afternoon}(h)\ \supset\ i = \mathsf{Fred} \tag{28}$$

then one instead can look for tuples of the form

$$\mathsf{G}' = \exists r \mathsf{Teaches}(\mathsf{Fred}, \mathsf{Math}, \mathsf{1pm}, r). \tag{29}$$

If the Teaches relation is indexed on its first argument, the constraint that $i = \mathsf{Fred}$ can result in a large speedup in enumerating solutions. If the first argument is the only argument indexed, then G requires scanning all tuples of the Teaches relation. G′, on the other hand, requires scanning only those tuples with Fred in the first position.

Even if other arguments of the Teaches relation are indexed, G′ can still be cheaper than G. For example, if the second argument is indexed, one would have to scan all Math classes to solve G, a set that would typically be much larger than the set scanned in solving G′, the set of classes that Fred teaches.

Query improvement via restrictions on variables is discussed thoroughly in King [44] and by U. S. Chakravarthy [12,14,13]. Note also that equality is not the only useful restriction that can be found. If the set of tuples for a given relation are sorted according to some ordering of one of its arguments (a "sorted index"), then finding some restriction on the range of values for this argument can also allow for a speedup. Again, see King [44] for many such examples.

3.6.4 Ramifications as Filters

Besides helping to *generate* fewer possibilties to test, knowing ramifications can reduce the expense of finding answers in by *filtering* partial answers. Consider a goal

$$\mathsf{G} = \exists x \exists y \mathsf{A}(x) \wedge \mathsf{B}(x, y).$$

In order to find the answers to such a goal, the set of x's for which $\mathsf{A}(x)$ holds is generated and then for each of these x's, the set of y's for which $\mathsf{B}(x,y)$ holds is generated.

Suppose now that some ramification $\mathsf{N} = \mathsf{N}(x)$ is derived for G. Consider the modified (and reordered) goal

$$\mathsf{G}' = \exists x \exists y \mathsf{A}(x) \wedge \mathsf{N}(x) \wedge \mathsf{B}(x,y).$$

If $\mathsf{N}(x)$ reduces the number of x's to consider, then one can avoid generating the set of possible y's for *each* of the x's eliminated from consideration. Note, of course, that the proper reordering of the conjuncts $\mathsf{A}(x)$, $\mathsf{B}(x,y)$, and $\mathsf{N}(x)$ is necessary to take advantage of any possible speedup.

Restating the above, let a is the number of x's for which $\mathsf{A}(x)$ holds and b be the average cost of finding all y's for which $\mathsf{B}(X,y)$ holds, where X is an arbitrary ground term. Then, without using $\mathsf{N}(x)$ as a filter, $O(ab)$ pairs of values must be considered for the ordered conjunction $\exists x \exists y \mathsf{A}(x) \wedge \mathsf{B}(x,y)$, that is

$$\mathsf{Cost}(\mathsf{Solving}(\mathsf{G})) = \mathsf{Cost}(\mathsf{Solving}(\mathsf{A}(x))) + ab. \tag{30}$$

Assume now that $\mathsf{N}(x)$ holds only for some fraction $\frac{1}{n}$ of the a answers for $\mathsf{A}(x)$. Then, for ordered conjunct $\mathsf{G}' = \mathsf{A}(x) \wedge \mathsf{N}(x) \wedge \mathsf{B}(x,y)$,

$$\begin{aligned} \mathsf{Cost}(\mathsf{Solving}(\mathsf{G}')) =& \\ &\mathsf{Cost}(\mathsf{Solving}(\mathsf{A}(x))) + \\ &a\mathsf{Cost}(\mathsf{Solving}(\mathsf{N}(X)) + \\ &\frac{ab}{n}, \end{aligned}$$

In other words, only $\frac{1}{n}$ of the pairs (x,y) that were considered for G need be considered for G$'$. By filtering one generated set first, the size of its cross product with another set has been reduced.

Example 3.10 Suppose we are looking for the set of radical staff members of presidents, that is,

$$\mathsf{G} = \exists x \exists y \mathsf{President}(x) \wedge \mathsf{OnTheStaff}(x,y) \wedge \mathsf{Radical}(y). \tag{31}$$

In order to find the pairs (x,y) satisfying this formula, we would have to generate the 40 x's for which $\mathsf{President}(x)$ hold, and then for each of these 40 presidents we must generate the say K people for whom $\mathsf{OnTheStaff}(x,y)$ holds, for some

y. Finally, Radical(y) is looked up on (x, y) pairs already generated. It generates no new pairs (although it may eliminate some old pairs), so in total, $40K$ pairs of values will be considered.

Lookup	Number of Values Generated
President(x)	40
OnTheStaff(x, y)	K
Radical(y)	1
Product	$40K$

(32)

Now, suppose **W** contains the proposition "Only a Democrat would have a radical on his staff," that is,

$$\forall\, u, v\ [\ \mathsf{Radical}(v) \wedge \mathsf{OnTheStaff}(u, v) \supset \mathsf{Democrat}(u)\ ]. \tag{33}$$

Since Democrat(x) is a ramification of G, consider the new ordered query

$$\mathsf{G}' = \exists x \exists y \mathsf{President}(x) \wedge \mathsf{Democrat}(x) \wedge \mathsf{OnTheStaff}(x, y) \wedge \mathsf{Radical}(y). \tag{34}$$

It is true that all 40 presidents must *still* be generated, *but only* 6 of the 40 presidents have been Democrats.[9] Thus, the ramification Democrat(x) filtered the set of 40 presidents to only 6 Democratic presidents. By doing the filtering before generating the cross product of x and y, we generate only $6K$ rather than $40K$ pairs as shown in Figure 35.

Lookup	Number of Values Generated
President(x)	40
Democrat(x)	$\frac{6}{40}$
OnTheStaff(x, y)	K
Radical(y)	1
Product	$6K$

(35)

Figure 9 illustrates how filtering works in this example. The presidents are along the x axis, and the vertical lines represent the staff members of each of the presidents. To solve G′, only the bold vertical lines are needed, whereas to solve G, all "40" vertical lines are needed.

[9] Woodrow Wilson, F. D. Roosevelt, Harry S. Truman, John F. Kennedy, Lyndon B. Johnson, and Jimmy Carter.

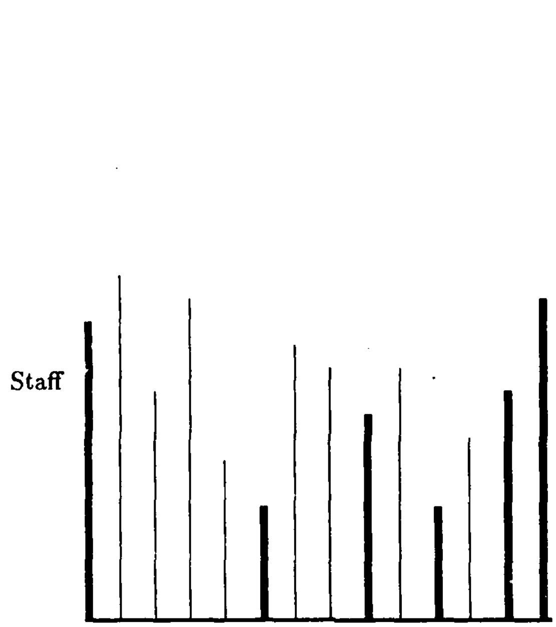


Figure 9: Using a Ramification as a Filter

Note also that the ramification Democrat(x) would be a terrible *generator* in this example. Since there are many, many more than 40 Democrats, trying to use the ramification Democrat(x) as a *generator* would have generated this huge set before using President(x) to reduce the set to only 6 Democratic presidents.

3.7 Summary

This chapter has defined *ramifications*, conditions that must accompany the achievement of a goal, and *supersumption*, the reformulation of a goal to include additional constraints such as ramifications. One purpose of finding and using ramifications is to widen the set of heuristics known to be applicable to a goal. Viewing goals as being partial descriptions of states, ramifications help complete the state's description thereby making a potentially larger set of heuristics directly applicable. Alternatively, ramifications, acting either as *filters* or *generators* of goal reductions, can reduce the size of the search space in which solutions might be found. As a filter a ramification is a formula that can be evaluated for a given goal, and if false, the goal can be immediately pruned. As a generator, the ramification is used as a source of potential goal reductions; if it is significantly cheaper to generate solutions via the ramification than via the original goal, an overall savings can result.

There has been a good deal of work related to supersumption, finding ramifications, and the mechanisms by which supersumptions of a goal can be cheaper to solve than the

original goal. The related work will be discussed that the end of Chapter 4, which discusses procedures for finding ramifications of a goal.

Chapter 4

Finding Ramifications

4.1 Introduction

Chapter 3 defined ramifications and showed how they can reduce the size of a search space. This chapter considers the problem of *generating* ramifications, that is,

> Given a database $\mathbf{W}$ of facts and a goal G, what procedure $\mathcal{P}$ will enumerate formulas N such that
> $$\mathbf{W} \models \forall x_1 \ldots \forall x_g (\widehat{\mathsf{G}} \supset \mathsf{N}), \quad (36)$$
> where G is assumed to be in prenex-conjunctive normal form[1] with matrix $\widehat{\mathsf{G}}$ a prefix $\square x_1 \ldots \square x_g$ and such that $\mathbf{y}_n \subseteq \{x_1, \ldots, x_g\}$, where $\mathbf{y}_N$ is the set of free variables of N.

For any such procedure $\mathcal{P}$, we must consider a number of factors. First, how *efficient* is $\mathcal{P}$? Second, is $\mathcal{P}$ ***sound and complete***, that is, for a given $\mathbf{W}$ and G, will $\mathcal{P}$ eventually generate all N for which (36) holds and none others?[2] Third, does $\mathcal{P}$ lend itself to *caching* its results for use in finding ramifications on other goals?

Four procedures will be considered for generating ramifications:

1. Lexical Generation of Ramifications ($\mathcal{P}_{Lex}$),
2. Natural Deduction on Subgoals ($\mathcal{P}_{Nat}$),
3. Resolution on Goal Clauses ($\mathcal{P}_{RGC}$), and

[1] Prenex-disjunctive normal form is also acceptable. See Manna [57] for presentation of prenex-conjunctive and prefix-disjunctive normal forms. The important result is that every sentence of first-order predicate calculus can be converted to a logically equivalent sentence in prenex-conjunctive (or in prenex-disjunctive) normal form in time proportional to the length of the formula.

[2] Actually, a slightly weaker notion (defined in Section 4.5) of completeness suffices and will be used here.

4. Resolution with Partial Subsumption ($\mathcal{P}_{RPS}$).

The first two methods, $\mathcal{P}_{Lex}$ and $\mathcal{P}_{Nat}$, are sound and complete, but terribly inefficient; they are briefly presented to show that soundness and completeness are easily achieved, but in and of themselves, these procedures are of little interest. The remaining two methods are based on binary resolution [76] under the set of support restriction [105]. The use of resolution in these two procedures differs from the usual use of resolution. Traditionally, resolution has been used as a *refutation* technique, that its, a technique for showing that a set of sentences is unsatisfiable. $\mathcal{P}_{RGS}$ and $\mathcal{P}_{RPS}$ use resolution as a *deduction* technique, that is, a way to find logical implications of a given sentence (Section 4.5 contains a discussion on the motivation for such usage of resolution.) $\mathcal{P}_{RGC}$ is a straightforward use of resolution as a forward inference technique. Soundness and completeness results are proven for $\mathcal{P}_{RGC}$ in Section 4.5. $\mathcal{P}_{RPS}$ is a less obvious and superior way in which to find ramifications. $\mathcal{P}_{RPS}$ derives ramifications via resolutions on the world model $\mathbf{W}$, but restricts the allowed resolutions via a restriction defined in terms of the goal G. $\mathcal{P}_{RPS}$ allows different goals to share their searches for ramifications, and even without such caching is identical in efficiency to $\mathcal{P}_{RGC}$.

Complexity of Finding Ramifications For arbitrary first-order formulas, the complexity of finding ramifications is bound by the limitation that the set of ramifications of a goal is recursively enumerable, but not recursive. A procedure such as $\mathcal{P}_{Lex}$ is such an enumeration procedure, but there is no procedure to decide in finite time whether an arbitrary formula is or is not a ramification of a given goal. It is easy to see this: If there were such a algorithm, one could decide whether a given formula N is a ramification of the goal "**true**", that is, whether or not

$$\mathbf{W} \models \mathsf{N},$$

a known impossibility. Thus, one can enumerate all ramifications, and one can determine in a finite (though unbounded) amount of time that a given formula is a ramification of a given goal and world model, but one cannot in general determine in a finite amount of time that a given formula is *not* a ramification of a given goal and world model.

4.2 Lexical Generation of Formulas ($\mathcal{P}_{Lex}$)

One sound and complete but very inefficient method of generating all ramifications of a goal is to systematically generate every possible well-formed formula (*wff*), checking each one to see if it is a ramification of the goal G. Procedure $\mathcal{P}_{Lex}$ is an outline of such a procedure.

(0) Procedure $\mathcal{P}_{Lex}$

(1) c $\leftarrow$ 0 (* number of candidate *wffs* so far *)

(2) for $l \leftarrow 1$ to ∞ do (* length of *wff* *)

(3) $\quad F_l \leftarrow \{ f \mid f$ is a *wff* of length l and contains free variables $\mathbf{x}_0 \subseteq \mathbf{x} \}$

(4) $\quad$ for each $f \in F_l$ do

(5) $\quad\quad B_c \leftarrow \mathbf{W} \cup \mathsf{CNF}(\neg(\forall x_1 \ldots \forall x_g\,(\widehat{\mathsf{G}} \supset f)))$

(6) $\quad\quad$ c $\leftarrow$ c+1

(7) $\quad$ for k $\leftarrow$ 0 to c do

(8) $\quad\quad r \leftarrow$ result of a resolution step on B_k (if possible)

(9) $\quad\quad$ if $r = \mathsf{NULL}$ then output(f_k)

It is easy to see that $\mathcal{P}_{Lex}$ is complete. There are a finite number (modulo variable names) of *wffs* of a given length and quite simple to find an algorithm for generating them all in a finite length of time. Such an algorithm is assumed in Step 3. Steps 4-6 set up the base set for checking (via resolution) whether a given *wff* f_c is a ramification of G. Finally, Steps 7-9 performs a single resolution step on every candidate *wff* f generated so far, outputting any f for which

$$\mathbf{W} \models \forall x_1 \ldots \forall x_g\,(\widehat{\mathsf{G}} \supset f). \tag{37}$$

Note that since a procedure checking a given f will not necessarily halt there must be interleaving of the checking of formulas of a given length with generation and checking of formulas of greater length. Procedure $\mathcal{P}_{Lex}$ guarantees that every candidate *wff* f eventually receives an unbounded number of resolution steps in attempting to prove (37). Thus it is guaranteed that for a given f, a proof of (37) will eventually be found if one exists.

Procedure $\mathcal{P}_{Lex}$, while sound and complete, is a very poor method of generating ramifications. Its failing is that it does not use the problem at hand in guiding its search for ramifications, nor does the procedure seem easily ammenable to such guidance. In order to make the procedure be responsive to the problem, the subgoal and known facts themselves must be used to guide the search for possible ramifications.

4.3 Natural Deduction on Subgoals ($\mathcal{P}_{Nat}$)

Probably the best known examples of deductively complete[3] systems for first-order predicate calculus are various natural deduction schemes, for example, the Gentzen system presented

[3] *Deductive* completeness is to be carefully distinguished from *refutation* completeness. A system is *deductively* complete if for any proposition entailed by a set of propositions, there exists a deduction of that

in Manna [57], page 108. Let us suppose a world model **W** consisting of a finite satisfiable set of closed well-formed formulas of first-order predicate calculus. Let G be a closed formula in prenex-conjunctive normal form with matrix $\widehat{\mathsf{G}}$ and prefix $\Box x_1 \ldots \Box X_g$.

The procedure $\mathcal{P}_{Nat}$ follows. Note that Step 2 is only partially specified. Even though the natural deduction system is complete, nothing has been stated about controlling the natural deduction steps. To insure that every possible ramification is eventually deduced, there must also be a control strategy guaranteeing that every possible proof will eventually be tried.

The $\mathcal{P}_{Nat}$ Procedure:

1. **Replace Variables** $x_1, \ldots, x_g$ **of** G via substitution σ with a set of new and distinct constants $X_1, \ldots, X_g$.
2. **Apply Natural Deduction Rules** to $\mathbf{W} \cup \{\widehat{\mathsf{G}}\sigma\}$
3. **Backsubstitute:** For any formula N′ deduced, return $\mathsf{N} = \forall y_1 \ldots \forall y_n (\mathsf{N}'\sigma^{-1})$, where $\{y_1, \ldots, y_n\}$ is the set of free variables in $\mathsf{N}'\sigma^{-1}$.[4]

The replacement of the variables $x_1 \ldots, x_g$ by new constants is needed so that the natural deduction will not deduce "ramifications" that are not true for all values of the variables of $\widehat{\mathsf{G}}$ in the formulas deduced. Since these constants are arbitrary, any formula N deduced is entailed for all values of $x_1, \ldots, x_g$. By soundness of the natural deduction system, for any formula N′ derived,

$$\mathbf{W}, \widehat{\mathsf{G}}\sigma \models \mathsf{N}$$

and by the Deduction Theorem,

$$\mathbf{W} \models (\widehat{\mathsf{G}}\sigma \supset \mathsf{N}).$$

The inverse substitution σ^{-1} can be applied to nz to regain the original free variables $x_1, \ldots, x_g$.

$\mathcal{P}_{Nat}$ is a complete method for finding ramifications of a subgoal. It is far better than $\mathcal{P}_{Lex}$ in that it is guided by the database **W** at hand. However, it is still lacking an essential feature - it is not sensitive to the actual goal at hand, that is, it will just as readily find implications of two random facts in the database as it will combine known facts with the goal in order to produce ramifications that depend upon the goal. But, The ramifications of most interest are those which depend upon the goal.

proposition. A system is *refutation* complete if "false" can be deduced from every unsatisfiable set of formulas. Resolution is refutation complete, but not deductively complete. For example, given a set of formulas $\{a, \neg a \vee b\}$, resolution cannot derive the formula $b \vee c$ in spite of the fact that $\{a, \neg a \vee b\}$ entails $b \vee c$.

[4] It is assumed, without loss of generality, that the set of variables of N′ and of G are disjoint. If not, the names of free variables of N′ must be changed.

In order to find ramifications that are consequences of the goal rather than the database alone, the search must be restricted to those formulas that depend upon the goal for support. Although we could constrain natural deduction to make only those deductions, it is easier to do so using resolution.[5]

4.4 Definitions for Resolution-Based Forward Reasoning

The $\mathcal{P}_{RGC}$ and $\mathcal{P}_{RPS}$ procedures use binary resolution [77] to generate ramifications. In presenting the two procedures, the definitions and terminology of this section will be useful.

In the rest of this chapter, it will be assumed that **W** consists of a finite set of clauses, each clause being implicitly universally quantified over each of its variables. Recall that a resolution refutation

Recall that a resolution refutation begins by creating the *base set*, that is, $\mathbf{W} \cup \mathsf{CNF}(\neg\mathsf{G})$, where $\mathsf{CNF}(f)$ is the set of clauses in the conjunctive normal form of f. It will be assumed that unless otherwise specified, all set of support deductions[6] have $\mathsf{CNF}(\neg\mathsf{G})$ as the initial set of support. Each clause C in the set of support is the *denial* of the conjunction $\mathsf{S} = \exists y_1 \ldots \exists y_c \neg\mathsf{C}$, where $\{y_1, \ldots, y_c\}$ is the set of variables in C. S can be viewed as a goal to which the original goal G has been reduced, that is, if we can find a solution to S we would also have a solution to the original goal G.

> **Example 4.1** Suppose $w_1 = \neg\mathsf{Zebra}(z) \vee \mathsf{Striped}(z) \in \mathbf{W}$ and $\mathsf{G} = \mathsf{Striped}(x)$. $\mathsf{CNF}(\neg\mathsf{G}) = \{\neg\mathsf{Striped}(x)\}$ and via resolution of this clause against w_1, the clause $\neg\mathsf{Zebra}(y)$ would be derived. $\neg\mathsf{Zebra}(y)$ is the denial of the subgoal $\exists y \mathsf{Zebra}(y)$ to which we have reduced the original goal G.

The **Extended World Model W*** is the set of all clauses from **W**, all clauses with all its parents in $\mathbf{W}^*$, and no other clauses. In other words, $\mathbf{W}^*$ is all clauses derived strictly from **W**. and as such, for all $w \in \mathbf{W}^*$, $\mathbf{W} \models w$.

The **Goal Set G*** includes $\mathsf{CNF}(\neg\mathsf{G})$, any clause with at least one parent in $\mathbf{G}^*$, and no other clauses. $\mathbf{G}^*$ is synonymous with the *set of support* obtained by starting with $\mathsf{CNF}(\neg\mathsf{G})$.

Given the above definitions, every deduction step on base set $\mathbf{W} \cup \mathsf{CNF}(\neg\mathsf{G})$ fits into exactly one of the following classes:

[5] Use of a natural deduction, as opposed to a resolution-based system, does not preclude using the goal in directing the search. See, for example, Boyer and Moore [7,6] or Bledsoe [4]

[6] See Section 2.4 for definitions of set of support and set of support deduction.

1. **WG Resolution** - a resolution between a clause of $\mathbf{G}^*$ and a clause of $\mathbf{W}^*$ yielding a new clause of $\mathbf{G}^*$.
2. **GG Resolution** - a resolution between two clauses of $\mathbf{G}^*$ yielding a new clause of $\mathbf{G}^*$.
3. **G Factoring** - factoring a clause of $\mathbf{G}^*$ yielding a new clause of $\mathbf{G}^*$.
4. **WW Resolution** - a resolution between two clauses of $\mathbf{W}^*$ yielding a new clause of $\mathbf{W}^*$.
5. **W Factoring** - factoring a clause of $\mathbf{W}^*$ yielding a new clause of $\mathbf{W}^*$.

Standard problem solving techniques can be viewed in terms of these resolution types. Backwards Reasoning (Goal Reduction) consists of GG and WG Resolution and G Factoring Steps. Forward Reasoning from the known facts of a problem consists of WW Resolution and WW Factoring Steps. Because most problems are such that goal reduction techniques search a smaller part of the space than forward reasoning from the known facts, we often restrict ourselves to backwards reasoning techniques such as the set of support strategy (of which backwards chaining is a special case). While generation of ramifications is a form of forward reasoning it differs from the common usage of forward reasoning in problem solving in a crucial way: *Rather than reason forward from the known facts for an entire problem, ramifications are generated by reasoning forward from goals created in the course of goal reduction.*

4.5 Resolution on Subgoal Clauses ($\mathcal{P}_{RGC}$)

Procedures $\mathcal{P}_{RGC}$ and $\mathcal{P}_{RPS}$ are based upon binary resolution [77]. Traditionally, resolution has been presented as a method for refutation of a set of clauses, that is, a proof that the set of clauses is unsatisfiable. Such a view is reinforced by resolution being *refutation* complete, but not *deductively* complete — resolution can deduce false from any unsatisfiable set of clauses, but cannot deduce all sentences (or even all clauses) entailed by a set of clauses.

In practice, however, resolution is very useful as a deduction rule as well. As was discussed in Chapter 2, goal-directed backwards reasoning can be performed via the set of support restriction on resolution. Further ordering of the allowed resolution steps gives depth-first, breadth-first or other search behavior. The main reason that resolution is useful in spite of its lack of deductive completeness is simple — the sentences entailed by the base set but not deducible using resolution are generally not of interest. As was seen in Chapter 2 and as will be seen in the following sections, if resolution cannot derive a proposition P,

it can derive a set of clauses which together are at least as useful as P. In backwards reasoning, useful will mean *weaker*, and in forward reasoning *stronger*.

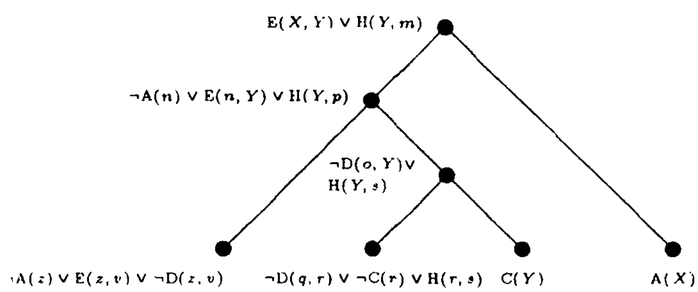
The RGC Procedure Let G be a closed formula in prenex-conjunctive normal form with matrix $\widehat{\mathsf{G}}$ and prefix $\Box x_1 \ldots \Box x_g$. Procedure $\mathcal{P}_{RGC}$ is as follows:

1. **Replace Variables** $x_1, \ldots, x_g$ **of** $\widehat{\mathsf{G}}$ via substitution σ with a set of new and distinct constants $X_1, \ldots, X_g$.
2. **Perform Set of Support Resolution**[7] on base set $\mathbf{W} \cup \{\widehat{\mathsf{G}}\sigma\}$ using the conjuncts of $\widehat{\mathsf{G}}\sigma$ as the initial set of support.
3. **Backsubstitute:** For any formula N′ deduced, return $\mathsf{N} = \forall y_1 \ldots \forall y_n(\mathsf{N}'\sigma^{-1})$, where $\{y_1, \ldots, y_n\}$ is the set of free variables in $\mathsf{N}'\sigma^{-1}$.[8]

Let us illustrate $\mathcal{P}_{RGC}$ with a simple example:

Example 4.2 Let $\mathsf{G} = \exists x \exists y(\mathsf{A}(x) \wedge \neg\mathsf{B}(x,y) \wedge \mathsf{C}(y))$ be the original goal, and let $\mathbf{W} = \{\neg\mathsf{A}(z) \vee \mathsf{E}(z,v) \vee \neg\mathsf{D}(z,v), \neg\mathsf{D}(q,r) \vee \neg\mathsf{C}(r) \vee \mathsf{H}(r,s), \mathsf{I}(t)\}$.

1. **Replace Variables:** Let $\theta = \{x \leftarrow X, y \leftarrow Y\}$ yielding $\widehat{\mathsf{G}}\theta = \mathsf{A}(X) \wedge \neg\mathsf{B}(X,Y) \wedge \mathsf{C}(Y)$
2. **Perform Set of Support Resolution** on base set $\mathbf{W} \cup \{\mathsf{A}(X), \neg\mathsf{B}(X,Y), \mathsf{C}(Y)\}$ and initial set of support $\{\mathsf{A}(X), \neg\mathsf{B}(X,Y), \mathsf{C}(Y)\}$. The resolution steps in the deduction tree below illustrate the derivation of the clause $N' = \mathsf{E}(X,Y) \vee \mathsf{H}(Y,m)$.



3. **Backsubstition** into $\mathsf{E}(X,Y) \vee \mathsf{H}(Y,m)$, yields the ramification $\mathsf{N} = \forall m \mathsf{E}(x,y) \vee \mathsf{H}(y,m)$.

[7] See Section 2.5.1 for definition of resolution, set of support, and deductions.

[8] It is assumed, without loss of generality, that the set of variables of N′ and of G are disjoint. If not, the names of free variables of N′ must be changed.

As was the case with $\mathcal{P}_{Nat}$ replacement of the variables of G's prefix prevents the resolution steps from deriving ramifications that are ramifications for *any* values of the free variables of G. For example, consider a goal $\exists x \mathsf{A}(x)$ and a world model $\mathbf{W}$ containing the clause $\neg \mathsf{A}(4) \vee \mathsf{D}(4)$. Without substituting some new ground variable for the x in $\mathsf{A}(x)$, one might think that $\mathsf{D}(4)$ is a ramification of $\mathsf{A}(x)$. Indeed it is not, since

$$\neg A(4) \vee \mathsf{D}(4) \not\models \forall x\, (\mathsf{A}(x) \supset \mathsf{D}(4)).$$

4.5.1 Soundness of $\mathcal{P}_{RGC}$

The soundness of $\mathcal{P}_{RGC}$ is an immediate consequence of the soundness of the resolution procedure.

Theorem 4.1 (Soundness of $\mathcal{P}_{RGC}$) *For any goal* G, *if* $\mathcal{P}_{RGC}$ *returns* N *then* N *is a ramification of* G.

Proof: $\mathcal{P}_{RGC}$ starts with base set $\mathbf{W} \cup \hat{\mathsf{G}}\sigma$. Note that since $\hat{\mathsf{G}}\sigma$ is a set of ground clauses, it is already in conjunct normal form. By the soundness of the resolution and factoring rules (See, for example, Theorem 5.1, page 72 of Chang and Lee [15]), we know that any clause N' derived via the resolution procedure is a logical consequence of the base set, i.e.,

$$\mathbf{W} \cup \hat{\mathsf{G}}\sigma \models \mathsf{N}', \tag{38}$$

or by the Deduction Theorem,

$$\mathbf{W} \models \hat{\mathsf{G}}\sigma \supset \mathsf{N}'. \tag{39}$$

Since σ is a ground substitution to arbitrary constants that do appear in $\mathbf{W}$,

$$\mathbf{W} \models \hat{\mathsf{G}}\sigma\sigma^{-1} \supset \mathsf{N}'\sigma^{-1},$$

and

$$\mathbf{W} \models \forall x_1 \ldots \forall x_g(\hat{\mathsf{G}} \supset \forall y_1 \ldots \forall y_n(\mathsf{N}'\sigma^{-1})),$$

or finally,

$$\mathbf{W} \models \forall x_1 \ldots \forall x_g(\hat{\mathsf{G}} \supset \mathsf{N}).$$

∎

4.5.2 Completeness of $\mathcal{P}_{RGC}$

The following theorem is the basic completeness result for $\mathcal{P}_{RGC}$:

Theorem 4.2 (Completeness of $\mathcal{P}_{RGC}$)[9] *Suppose we are given*

1. $\mathbf{W}$*, a satisfiable set of clauses, implicitly universally quantified*
2. $\mathbf{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_g\}$*, a set of ground literals, and*
3. $\mathsf{N} = \mathsf{n}_1 \vee \ldots \vee \mathsf{n}_n$*, a clause with variables* $\{y_1, \ldots, y_m\}$

such that $\mathbf{W} \models (\mathbf{g}_1 \wedge \ldots \wedge \mathbf{g}_g) \supset \forall y_1 \ldots \forall y_m \mathsf{N}$ *and such that* $\mathbf{W} \not\models \mathsf{N}$*, then there exists a set of support deduction of clause* M *from base set* $\mathbf{W} \cup \mathbf{G}$ *with initial set of support* $\mathbf{G}$ *such that* M *subsumes* N*.*

Proof Since $\mathbf{W} \models (\mathbf{g}_1 \wedge \ldots \wedge \mathbf{g}_g) \supset \forall y_1 \ldots \forall y_m \mathsf{N}$, then for new and distinct constants $Y_1, \ldots, Y_m$ and substitution $\theta = \{y_1 \rightarrow Y_1, \ldots, y_m \rightarrow Y_m\}$, it must be that $\mathbf{W} \cup \mathbf{G} \models \mathsf{N}\theta$. Thus, the set $\mathbf{W} \cup \mathbf{G} \cup \{\neg \mathsf{n}_1\theta, \ldots \neg \mathsf{n}_m\theta\}$ is unsatisfiable, and by Herbrand's Theorem, there exists a finite set $\mathbf{C}$ of ground instances of the above set that is unsatisfiable. Let $\mathbf{C}_w$ be the set of ground instances of clause of $\mathbf{W}$ that are in $\mathbf{C}$, and similarly define $\mathbf{C}_g \subseteq \mathbf{G}$ and $\mathbf{C}_n \subseteq \{\neg \mathsf{n}_1\theta, \ldots \neg \mathsf{n}_m\theta\}$. By hypothesis, $\mathbf{W} \not\models \mathsf{N}$ and therefore $\mathbf{W} \not\models \mathsf{N}\theta$, since the Y_i are arbitrary. Thus $\mathbf{C}_w \cup \mathbf{C}_n$ is satisfiable, and so there must exist a set of support refutation of $\mathbf{C}$ from initial set of support $\mathbf{C}_g$. By Lemmas 2.12 and 2.13, this refutation can be converted to a set of support deduction of a clause M' with base set $\mathbf{C}_w \cup \mathbf{C}_g$, initial set of support $\mathbf{C}_g$, and such that $\mathsf{M}' \subseteq \mathsf{N}\theta$. We now use the Lifing Lemma to build another deduction in which the clauses from $\mathbf{C}_w$ are replaced by the corresponding clauses from $\mathbf{W}$. This builds a set of support deduction of a clause M from $\mathbf{W} \cup \mathbf{G}$ and initial set of support $\mathbf{G}$, and such that $\mathsf{M}\rho = \mathsf{M}'$ for some ρ. So, we have $\mathsf{M}\rho \subseteq \mathsf{N}\theta$, but since θ is invertible (i.e., the Y_i are distinct), $\mathsf{M}\phi \subseteq \mathsf{N}$, where $\phi = \rho\theta^{-1}$. Thus, there exists a set of support deduction of a clause M from base set $\mathbf{W} \cup \mathbf{G}$, initial set of support $\mathbf{G}$, and such that M subsumes N. ∎

The above result does not say that $\mathcal{P}_{RGC}$ can deduce any ramification of any conjunctive goal. Instead, it says that if it cannot deduce that ramification, it will deduce one that is at least as powerful. Before discussing this, a word on clausal form is in order:

[9] This theorem, though found independently, is a variation of theorems proven by Lee [50] and Minicozzi and Reiter [70]. See Section 4.8.8.

An arbitrary formula is not necessarily equivalent to its conjunct normal-form. The lack of equivalence arises due to Skolemization.

> **Example 4.3** $\mathsf{A}(\mathsf{K})$ is a conjunct normal-form for the formula $\exists x \mathsf{A}(x)$, where K is a Skolem constant. Suppose the domain is the set of natural numbers and $\mathsf{A}(0)$ is valid, but $\neg\mathsf{A}(y)$ for all $y > 0$. Then $\mathsf{A}(\mathsf{K})$ is false under any interpretation in which K is assigned a non-zero constant, but $\exists x \mathsf{A}(x)$ can still be true under such an interpretation.

Note also that the conjunct normal-form for a given formula is not unique — one can choose any new name for Skolem functions.

What Theorem 4.2 says is that for some way of converting an arbitrary formula N to conjunct normal-form, $\mathcal{P}_{RGC}$ can derive a set of clauses N' such that each clause in the conjunct normal-form of N is subsumed by a clause of N'.

One might question whether finding a subsuming clause is of value. After all, the original goal itself is "at least as powerful" as the ramification in the sense that the necessasry constraint can be derived from it. It turns out, however, that subsumption is just the desired relationship between an arbitrary clause N and a clause N' derived by $\mathcal{P}_{RGC}$. The subsumed clause N can differ from N' only in (1) having more disjuncts (literals), and (2) having some variables in N' replaced by constrants in N. It immediately follows that the subsumed clause N is a ramification if N' is a ramification. It is preferable to know that A is a ramification of G rather than that $A \vee B$ is a ramification. Similarly, it is preferable to know that $\forall y A(y)$ is a ramification of G rather than to know only that $A(4)$ is a ramification of G. Stated differently and only slightly inaccurately, $\mathcal{P}_{RGC}$ *does can* find arbitrary ramifications; it simply eliminates unnecessary variable bindings and unnecessary disjuncts.

4.5.3 Caching the Results of $\mathcal{P}_{RGC}$

In performing goal reduction on some goal G, it is likely that one will encounter many similar subgoals. In terms of resolution, many clauses in the set of support will have sets of literals in common.[10] As a result some of the ramifications of one subgoal clause will often be the same[11] as the ramifications of other clauses; it would be wasteful to generate these ramifications from scratch for each similar subgoal clause. In addition, it is usually impossible to compute *all* the ramifications of a subgoal. For a given subgoal clause it is

[10] Modulo variable names

[11] Again modulo variable names

desirable to store the results to date in order to begin the search anew sometime later. As will be seen in Section 4.7, a third reason to cache ramifications is that subgoals of a goal usually inherit all the ramifications of that goal.

Fortunately, there is a straightforward way to cache such results, namely, in **W** itself. By definition, if clause N is a ramification of conjunction $\mathsf{G} = \mathsf{G}_1 \wedge \ldots \wedge \mathsf{G}_g$, then

$$\mathbf{W} \models \neg \mathsf{G}_1 \vee \ldots \vee \neg \mathsf{G}_g \vee \mathsf{N},$$

and so $\neg \mathsf{G}_1 \vee \ldots \vee \neg \mathsf{G}_g \vee \mathsf{N}$ can be added to **W**.

If the ramification follows from a proper subset G′ of the conjuncts of G it is preferable to note this fact in **W** via a clause $\neg \mathsf{G}' \vee \mathsf{N}$ instead of $\neg \mathsf{G} \vee \mathsf{N}$.

et us refer to G′ as a *foundation* of N, that is,

Definition 4.1 *For a conjunctive goal* G *and a clause* N, *where* N *is a ramification of* G, *a a* foundation *of* N *is any subset* G′ *of the conjunction* G *such that* $\mathbf{W} \models \neg \mathsf{G}' \vee \mathsf{N}$.

One would like to require a foundation to be minimal, that is, that no subset of a foundation is also a foundation of that ramification. Unfortunately, the problem of determining whether a given foundation is minimal is semi-decidable, and minimality will not be required in the discussion that follows.

A given deduction of a ramification N from a goal G may not involve all of the conjuncts of G. While minimality is too strong a condition to require, it is easy to note only the conjuncts of G that were actually needed in the given deduction of G. To do so, $\mathcal{P}_{RGC}$ can be modified to record such a foundation of each ramification deduced. Let us notate the foundation of a clause i as F_i. The modified procedure is:

1. **Replace Variables** $x_1, \ldots, x_g$ **of** $\widehat{\mathsf{G}}$ via substitution σ with a set of new and distinct constants $X_1, \ldots, X_g$.

2. **Record Initial Foundations** of clauses from **W** and literals of $\widehat{\mathsf{G}}\sigma$. For every clause w of **W**, $F_w = \{\}$. For every literal g in $\widehat{\mathsf{G}}\sigma$, $F_g = g$.

3. **Perform Set of Support Resolution** on base set $\mathbf{W} \cup \{\widehat{\mathsf{G}}\sigma\}$ using the conjuncts of $\widehat{\mathsf{G}}\sigma$ as the initial set of support. For each clause n deduced, if n has only one parent p, then $F_n = F_p$. If n has parents p and q, then $F_n = F_p \cup F_q$.

4. **Backsubstitute:** For any formula N′ deduced, return $\mathsf{N} = \forall y_1 \ldots \forall y_n(\mathsf{N}'\sigma^{-1})$, where $\{y_1, \ldots, y_n\}$ is the set of free variables in $\mathsf{N}'\sigma^{-1}$. In addition, the clause $((\bigvee \neg F_{N'}) \vee \mathsf{N})\varphi$ may be added to **W**, where φ is a uniform renaming of the variables in $(\bigvee \neg F_{N'}) \vee \mathsf{N}$.

Example 4.4 Suppose we are given

$$\hat{\mathsf{G}} = \{\mathsf{A}(y), \mathsf{B}(y,z)\}$$
$$w_1 = \neg\mathsf{A}(m) \vee \mathsf{C}(m,n)$$
$$w_2 = \neg\mathsf{C}(o,p) \vee \mathsf{D}(o)$$
$$w_1, w_2 \in \mathbf{W}.$$

The substitution $\{y \to Y, z \to Z\}$ is used to replace the variables of G. When $\mathcal{P}_{RGC}$ generates $N_1 = \mathsf{C}(Y,n)$, F_{N_1} is $\{\mathsf{A}(Y)\}$ so $\neg\mathsf{A}(q) \vee \mathsf{C}(q,r)$ can be added to $\mathbf{W}$, where q and r are new variables. Similarly, upon generation of $N_2 = \mathsf{D}(Y)$, $\neg\mathsf{A}(s) \vee \mathsf{D}(s)$ can be added to $\mathbf{W}$.

4.6 Resolution with Partial Subsumption ($\mathcal{P}_{RPS}$)

The $\mathcal{P}_{RGC}$ Procedure of Section 4.5 takes a conjunction of literals $\mathsf{G} = \mathsf{g}_1 \wedge \ldots \wedge \mathsf{g}_g$ and reasons forward from it in order to generate ramifications. Unfortunately, G does not appear in the normal course of resolution (or resolution-based residue techniques); its negation,[12] the clause $\mathsf{C} = \neg\mathsf{G}$ appears in the set of support. In order to use $\mathcal{P}_{RGC}$ a new base set $B_{RGC} = \mathbf{W} \cup \neg\mathsf{C}\sigma$ must be constructed (where σ is a substitution replacing variables of G with new and distinct constants) on which to perform resolution.

By a slight modification of $\mathcal{P}_{RGC}$ backward reasoning (goal reduction) and forward reasoning (generation of ramifications) can be performed without re-negation of goal clauses and using the same resolution inference engine. The modified technique will be called "Resolution with Partial Subsumption"[13] or $\mathcal{P}_{RPS}$.

$\mathcal{P}_{RPS}$ is a severely restriction on set of WW-Resolution and W-Factoring steps, generating the same set of ramifications ramifications as were generated by $\mathcal{P}_{RGC}$. Instead of directly generating the ramifications as does $\mathcal{P}_{RGC}$, $\mathcal{P}_{RPS}$ generates clauses of the form

$$\mathsf{P} = \mathsf{c}_1 \vee \ldots \vee \mathsf{c}_m \vee \mathsf{n}_1 \vee \ldots \vee \mathsf{n}_n,$$

where $\mathsf{P}_C = \mathsf{c}_1 \vee \ldots \vee \mathsf{c}_m$ subsumes the negation of a G and $\mathsf{P}_N = \mathsf{n}_1 \vee \ldots \vee \mathsf{n}_m$ is a ramification of G. As stated above, if resolution is being used for goal reduction (as in Resolution Residue), the conjunct G will not explicitly appear, but rather its negation.

[12] More accurately, the disjunction of the complement of each of its conjuncts.

[13] The name "Partial Subsumption" has been used to show the similarity to Chakravarthy's use of the same term (Chakravarthy [12,14,13]).

Example 4.5 Suppose **W** contains clauses $\neg A \vee B$ and $\neg B \vee C$. Given $G = A \wedge Z$, $\mathcal{P}_{RGC}$ it negates the clause $\neg A \vee \neg Z$ that actually appeared and derives the ramification C. In contrast, $\mathcal{P}_{RPS}$ simply resolves the two clauses of $\mathbf{W}^*$ together producing a new clause $\neg A \vee C$ stating that C is a ramification of any subgoal containing the conjunct A.

The above scheme has the advantage that the same database can be used for both forward and backward reasoning. In addition, if the results of forward reasoning happen to apply to more than one subgoal, they will already be present without any explicit caching mechanism. Let us now explore $\mathcal{P}_{RPS}$ in more detail.

4.6.1 The $\mathcal{P}_{RPS}$ Procedure

Suppose that **W** is a satisfiable set of clauses, and that there exists a goal $G = G_1 \wedge \ldots \wedge G_m$, where the G_i are literals. In other words, $\neg G \in \mathbf{G}^*$. Let σ be a substitution replacing each variable of G with a new and distinct constant.

The $\mathcal{P}_{RPS}$ procedure is stated as the following restriction on resolution as follows:

> For $C_G \in \mathbf{G}^*$, $\mathcal{P}_{RPS}$ starts with base set **W** and allows any resolution (or factoring) steps such that the resolvent (factor) contains a literal L that unifies with a literal of $C_G\sigma$.

Note that it is impossible that a resolvent (or factor) C have a literal that unifies with a literal of C_G unless the same is true for at least one parent of C.

Example 4.6 Given goal $G = A(x) \wedge B(x) \wedge C(x)$, $\sigma = \{x \rightarrow X\}$ and $w_1, \ldots, w_5 \in \mathbf{W}$, where

$$
\begin{aligned}
w_1 &= \neg A(z) \vee D(z) \\
w_2 &= \neg D(y) \vee E(y) \\
w_3 &= \neg D(4) \vee F(4) \\
w_4 &= \neg A(u) \vee \neg B(u) \vee H(u) \\
w_5 &= \neg I(v) \vee A(v),
\end{aligned}
$$

$\mathcal{P}_{RPS}$ can resolve w_1 and w_2 and add the resolvent $w_6 = \neg A(t) \vee E(t)$.

$\mathcal{P}_{RPS}$ can not resolve w_1 and w_3 because the resolvent $\neg\mathsf{A}(4) \vee \mathsf{F}(4)$ has no literals that unify with a literal in $\neg\mathsf{G}\sigma$.

$\mathcal{P}_{RPS}$ can not resolve w_1 and w_5 because the resolvent $\neg\mathsf{I}(s) \vee \mathsf{D}(s)$ has no literals that unify with a literal in $\neg\mathsf{G}\sigma$.

$\mathcal{P}_{RPS}$ can resolve w_4 and w_5 — the resolvent $\neg\mathsf{I}(r) \vee \neg\mathsf{B}(r) \vee \mathsf{H}(r)$ has a literal that unifies with a literal in $\neg\mathsf{G}\sigma$.

It is tempting to try to use a stronger restriction, namely that if a parent has a literal that unifies with a literal in C_G, then the resolvent (factor) must also have a literal that unifies with that literal of C_G. It turns out that this is too strong a restriction for completeness.

4.6.2 Soundness of $\mathcal{P}_{RPS}$

$\mathcal{P}_{RPS}$ is sound in that each clause deduced is the proof of a given ramification.

Theorem 4.3 (Soundness of $\mathcal{P}_{RPS}$) *Suppose*

1. $\mathbf{W}$ *is a satisfiable set of clauses,*
2. $\mathsf{G}' = \neg g_1 \vee \ldots \vee \neg g_g \in \mathbf{G}^*$
3. $C = c_1 \vee \ldots \vee c_q$ *is a clause deduced by* $\mathcal{P}_{RPS}$.

Then there exists a substitution θ *and some non-empty* $C' \subseteq C$ *such that* $C'\theta \subseteq \mathsf{G}'$ *and*

$$\mathbf{W} \models (\bigwedge(\neg C') \supset \bigvee(C - C').$$

Proof: The proof is quite straightforward. Let σ be the substitution replacing the variables of G by new and distinct constants. Since $\mathcal{P}_{RPS}$ performs resolution on base set $\mathbf{W}$, any clause C generated by $\mathcal{P}_{RPS}$ is such that $\mathbf{W} \models C$. So, for any $C' \subseteq C$,

$$\mathbf{W} \models (\bigwedge(\neg C') \supset \bigvee(C - C').$$

Furthermore, since every clause C generated by $\mathcal{P}_{RPS}$ contains at least one literal that unifies with a literal from $\mathsf{G}'\sigma$ it is guaranteed that there is some substitution θ such that at least one of the literals in $C\theta$ is a literal of G'. Thus, C' need not be empty. ∎

4.6.3 Completeness of $\mathcal{P}_{RPS}$

The basic completeness result for $\mathcal{P}_{RPS}$ is similar to that for $\mathcal{P}_{RGC}$ — not every ramification N can be deduced, but ramifications that subsume every clause in a conjunct normal form of N *are* deducible by $\mathcal{P}_{RPS}$. In addition, the deduced clauses tell which conjuncts of G were needed for the deduction.

The completeness result for $\mathcal{P}_{RPS}$ is Theorem 4.7, but prior to proving it, some preliminary results are needed.

Lemma 4.4 (Bubble Lemma) *Suppose that D' is a binary resolvent of clauses C_1' and C_2'. Suppose further that $C_1 = C_1'\phi_1^{-1} \cup M$ and $C_2 = C_2'\phi_2^{-1} \cup N$ for some substitution ϕ_1 and ϕ_2, and sets of literals M and N. Then C_1 and C_2 (or factors of C_1 and/or C_2) have a binary resolvent D such that for some θ, $D' \subseteq D\theta$, and $D\theta - D' \subseteq (M \cup N)\theta$.*

Proof By the Lifting Lemma (Lemma 2.4), the clauses $C_1\phi_1^{-1}$ and $C_2\phi_2^{-1}$ (or facotrs of these clauses) have a resolvent Q such that D' is an instance of Q, that is, for some substitution θ, $Q\theta = D'$. The addition of literals M to clauses $C_1\phi_1^{-1}$ and N to $C_2\phi_2^{-1}$ does not change the fact that θ is still an mgu for literals in $C_1\phi_1^{-1}$ and $C_2\phi_2^{-1}$, and thus clauses C_1 and C_2 (or factors of these clauses) must also have a resolvent D via mgu θ. All literals present in Q will also be present in D, so $D' \subseteq D\theta$. The additional literals of D will be from $(M \cup N)\theta$, so $D\theta - D' \subseteq (M \cup N)\theta$. Note that some of the literals of $(M \cup N)\theta$ might be identical to literals of Q, and thus it is not correct to say $(D - (M \cup N))\theta = D'$.
∎

Lemma 4.5 (Bubble Lemma (Factoring)) *Suppose that D' is a factor of clause C'. Suppose further that clause $C = C'\varphi^{-1} \cup M$ for some substitution φ, and a set of literals M. Then C has a factor D such that for some θ, $D' \subseteq D\theta$, and $D\theta - D' \subseteq M\theta$.*

Proof By hypothesis, there exists some substitution ρ, a most general unifier of two or more literals of C'. Since $(C - M)\varphi = C'$, it must be the case that the substitution $\varphi\rho$ unifies two or more literals of C, and thus $D' \subseteq D\varphi\rho$ and $D\varphi\rho - D' \subseteq M\varphi\rho$. Even if $\varphi\rho$ is not an mgu of the subset of literals from C, the existence of $\varphi\rho$ implies the existence of an mgu θ with the above properties.
∎

Theorem 4.6 (Bubble Theorem) *Let* M *be a satisfiable set of clauses and* U *be a satisfiable set of unit clauses such that there exists a set of support deduction D of a clause N'*

from base set $\mathbf{M} \cup \mathbf{U}$ *and initial set of support* $\mathbf{U}$. *Let* $\mathbf{S}$ *be the smallest subset of* $\mathbf{M}$ *containing all clauses with a literal* L *such that* $\neg L$ *unifies with some* $u \in \mathbf{U}$. *Then there exists a set of support deduction of a clause* $C = N \cup P$ *from base set* $\mathbf{M}$ *and initial set of support* $\mathbf{S}$, *where*

1. N' *is an instance of* N, *and*
2. P *subsumes the clause* $P' = \bigvee \neg u$ *for all* $u \in \mathbf{U}$.

Proof: Since $\mathbf{U}$ is satisfiable, no two clauses of $\mathbf{U}$ can resolve against each other. Therefore every clause from $\mathbf{U}$ in the fringe of D is resolved against a clause of $\mathbf{M}$. Consider a new deduction tree D' with a fringe $\mathsf{Fringe}(D') = \mathsf{Fringe}(D) - \mathbf{U}$. For every subtree of D, there will be a corresponding deduction, i.e., subtree of D' for which the theorem holds. The proof is by induction on the height n of the subtrees of deduction tree D'.

Base Case ($n = 1$): Since D is a set of support deduction using nodes from $\mathbf{U}$ as the initial set of support, every clause m in the fringe of D' is a clause that resolved against a clause of $\mathbf{U}$ in D. Thus the clause m contains a literals whose negation unifies with some $u \in \mathbf{U}$. In addition, $m = n \cup p$, where p is a singleton set that subsumes P' (in other words, p contains the literal that resolved against the unit clause of $\mathbf{U}$), and $n\theta = r$, where r is the resolvent of m and a clause from $\mathbf{U}$ in D.

Induction Step ($n = k$): Assume that the theorem holds for all subtrees of height $k - 1$ or less. Each subtree of height k was created either via a binary resolution or via a factoring step. If the step was a binary resolution step, by Lemma 4.4, there exists a resolvent $C = N \cup P$ such that (1) the corresponding node in D is an instance of N and (2) there exists a substitution σ such that $P\sigma \subseteq P'$. Since by hypothesis at least one of the parent clauses of C was in the set of support, then so is C. Simlarly, if the step was a factoring step, then by Lemma 4.5 and a similar argument, the factor is in the set of suppport, the corresponding node of D is an instance of some of the literals of the clause, and the remaining literals subsume P'. ∎

The major result of this section can now be stated and proven:

Theorem 4.7 (Completeness of $\mathcal{P}_{RPS}$) *Suppose*

1. $\mathbf{W}$ *is a satisfiable set of clauses,*
2. $\mathsf{G} = \mathsf{g}_1 \wedge \ldots, \wedge \mathsf{g}_g$ *is a conjunction of literals, such that* $\mathbf{W} \cup \{\mathsf{G}\}$ *is satisfiable,*

3. $\mathsf{N} = \mathsf{n}_1 \vee \ldots \vee \mathsf{n}_n$ *is a clause and a ramification of* G *given* $\mathbf{W}$*, and such that* $\mathbf{W} \not\models \mathsf{N}$.

There exists a set of support deduction D of a clause $M = M_G \cup M_N$ from base set $\mathbf{W}$ *and initial set of support* $\mathbf{S}$*, where*

1. *M_G subsumes* $\neg$G*,*
2. *M_N subsumes* N*, and*
3. $\mathbf{S}$ *contains exactly those clauses of* $\mathbf{W}$ *having a literal l such that $\neg l$ unifies with some* $\mathbf{g}_i$.
4. *Every non-fringe clause in D contains at least one literal l such that $\neg l$ unifies with some* $\mathbf{g}_i$.

Proof:

Let $\sigma_g = \{x_1 \rightarrow X_1, \ldots, x_g \rightarrow X_g\}$, where $\{x_1, \ldots, x_g\}$ is the set of variables in G, and $\{X_1, \ldots, X_g\}$ is a set of new and distinct constants.
Let $\sigma_n = \{y_1 \rightarrow Y_1, \ldots, y_n \rightarrow Y_n\}$, where $\{y_1, \ldots, y_n\}$ is the set of variables in $\mathsf{N}\sigma_g$, and $\{Y_1, \ldots, Y_n\}$ is a set of new and distinct constants.

Let $\mathbf{G} = \{\mathsf{g}_1\sigma_g, \ldots, \mathsf{g}_g\sigma_g\}$ and let $\mathbf{N}' = \{\neg\mathsf{n}_1\sigma_g\sigma_n, \ldots, \neg\mathsf{n}_n\sigma_g\sigma_n\}$.

Since N is a ramification of G, and because the X_i and Y_i are distinct and arbitrary,

$$\mathbf{W} \models (\mathsf{g}_1\sigma_g \wedge \ldots \wedge \mathsf{g}_g\sigma_g) \supset \mathsf{N}\sigma_g\sigma_n,$$

and therefore the set $\mathbf{W} \cup \mathbf{G} \cup \mathbf{N}'$ is unsatisfiable. By Herbrand's Theorem, there must exist a finite set $\mathbf{H}$ of ground instances of $\mathbf{W} \cup \mathbf{G} \cup \mathbf{N}'$ that is also unsatisfiable. Since $\mathbf{W} \cup \mathbf{N}'$ and $\mathbf{W} \cup \mathbf{G}$ are both satisfiable, $\mathbf{H}$ must contain clauses both from $\mathbf{G}$ and $\mathbf{N}'$. Let the set of clauses in $\mathbf{H}$ from $\mathbf{W}$, $\mathbf{G}$ and $\mathbf{N}'$ be notated as $\widehat{\mathbf{W}}$, $\widehat{\mathbf{G}}$ and $\widehat{\mathbf{N}'}$, respectively.

Constuction 0: Let $\widehat{\mathbf{W}_0}$ be the subset of $\widehat{\mathbf{W}}$ constructed by removing from $\widehat{\mathbf{W}}$ every clause $g \cup m$, where $g \in \widehat{\mathbf{G}}$, and let $\mathbf{H}_0 = \widehat{\mathbf{W}_0} \cup \widehat{\mathbf{G}} \cup \widehat{\mathbf{N}'}$. Since $\mathbf{H}_0$ contains all the unit ground clauses $\mathbf{g} \in \widehat{\mathbf{G}}$, $\mathbf{H}_0 \models \mathbf{H}$, and $\mathbf{H}_0$ is also unsatisfiable. Since $\mathbf{W} \cup \mathbf{N}'$ is satisfiable, so is $\widehat{\mathbf{W}_0} \cup \widehat{\mathbf{N}'}$, and so by Theorem 4.2, there must then be a set of support deduction D_0 of a clause N_0 from base set $\widehat{\mathbf{W}_0} \cup \widehat{\mathbf{G}}$ and initial set of support $\widehat{\mathbf{G}}$ such that N_0 subsumes (is a subset of) $\mathsf{N}\sigma_g\sigma_n$.

Constuction 1: Let $\widehat{\mathbf{W}_{0G}}$ be the subset of $\widehat{\mathbf{W}_0}$ containing all clauses that contain a literal l such that $\neg l \in \widehat{\mathbf{G}}$. Based on deduction D_0 and Theorem 4.6, there also exists a set of support deduction D_1 of a clause $\mathsf{C}_1 = \mathsf{P}_1 \cup \mathsf{N}_1$ from base set $\widehat{\mathbf{W}_0}$ and initial set of support $\widehat{\mathbf{W}_{0G}}$, where N_1 subsumes N_0 (and therefore subsumes $\mathsf{N}\sigma_g\sigma_n$) and P_1 subsumes $\bigvee_i \neg\mathsf{g}_i$, for all $\mathsf{g}_i \in \widehat{\mathbf{G}}$. Furthermore,

since $\widehat{\mathbf{W}_0}$ contains no clauses with a literal $l \in \widehat{\mathbf{G}}$, every non-fringe clause of D_1 is guaranteed to contain a literal l such that $\neg l \in \widehat{\mathbf{G}}$.

Lifting: The Lifting Lemma can be used to turn Deduction D_1 into the desired Deduction D. Each node in the fringe of Deduction D_1 can be replaced by the corresponding clause in $\mathbf{W}$. By induction on subtrees, the Lifting Lemma can be shown to lift these variables to the root, deriving a clause M. Since each clause of D_1 contains a literal l such that $\neg l \in \widehat{\mathbf{G}}$, each non-fringe clause of D contains a literal whose complement unifies with a clause of $\widehat{\mathbf{G}}$. Furthermore, since the X_i do not appear in $\mathbf{W}$, each non-fringe clause of D contains a literal l whose complement unifies with a conjunct of G.

By Theorem 4.6 it is also the case that N_1 is an instance of M_N, that is, there exists a θ such that

$$M_N\theta = \mathsf{N}_1 \subseteq \mathsf{N}\sigma_g\sigma_n.$$

But since $(\sigma_g\sigma_n)$ is invertible,

$$M_N\theta(\sigma_g\sigma_n)^{-1} \subseteq \mathsf{N},$$

in other words, M_N subsumes N. ∎

4.7 Inheritance of Ramifications

So far, deduction of ramifications of a single goal has been considered. In practice, however, finding ramifications and goal reduction via backwards reasoning are interleaved. Instead of just wanting to know ramifications of a single goal, it would also be useful to know whether ramifications of a goal G_1 are still valid for goals further down in the backwards-reasoning deduction tree. If so, any work done in finding ramifications for a goal G_1 need not be repeated to find the same ramification for goal G_2. Furthermore, to find additional ramifications for G_2, forward reasoning could start from the inherited ramification rather than from scratch.

Although it seems reasonable that ramifications can be inherited, *it is not always the case that a ramification of one goal is a ramification of its offspring.* One reason is the renaming and binding of variables, but that is easily taken care of. The other reason is due to somewhat pathological cases involving the merging of two goals (GG-Resolutions, in the terminology of Section 4.4). In this section, a precise formulation is given for what ramifications may be gleaned from ramifications of an ancestor goal.

Usually, ramifications are inherited from parent nodes (modulo variable substitutions). It is easy to see why this is so: Suppose a goal G_1 gives rise to a new goal G_2 via backward reasoning. Suppose also that G_1 has ramification N. Since G_2 was derived from G_1 via backwards reasoning, then G_1 could be derived from G_2 by forward reasoning. But, since N was derived from G_1 via forward reasoning as well,

$$G_2 \xrightarrow{FR} G_1 \xrightarrow{FR} N,$$

that is, N should be derivable from both G_1 and G_2 by forward reasoning.

Example 4.7 Consider a goal $G_1 = C \wedge D$. If $C \supset N$, then N is a ramification on G_1. Suppose now that $A \wedge B \supset C$ is applied to reduce G_1 to $G_2 = A \wedge B \wedge D$. Clearly, since $A \wedge B \supset C$ and $C \supset N$, then $A \wedge B \supset N$, and so the ramification N is inherited by G_2.

Let us now consider inheritance of ramifications for clauses in $\mathbf{G}^*$ for various possible steps of a resolution refutation (See Section 4.4 for explanation of terms $\mathbf{G}^*$, $\mathbf{W}^*$, WG Resolution, GG Resolution, etc.). Consider a clause $C_G \in \mathbf{G}^*$, that is, clauses from the negation of the original goal G has been reduced via 0 or more resolution and factoring steps to a clause C_G. Let $G_1(y) = \neg C_G$. Suppose also that G_1 has some ramification N. If G_1 is further reduced to some new goal G_2, will N also be a ramification (modulo variable names) of G_2? The next three subsections address that issue with a series of theorems which follow easily from the definitions of resolution and ramification.

4.7.1 Inheritance under WG-Resolution Steps

The following theorem holds for WG Resolution steps:

Theorem 4.8 *Suppose C_G is a clause from $\mathbf{G}^*$, C_W is a clause from $\mathbf{W}^*$, and C_R is a resolvent of C_G and C_W via unifier σ. If N is a ramification of $\neg C_G$ then $N\sigma$ is a ramification of C_R.*

Proof: Let us represent the various clauses as follows:

$$C_G = \neg L_{G1} \vee \ldots \vee \neg L_{Gn} \tag{40}$$

$$C_W = \neg L_{W1} \vee \ldots \vee \neg L_{Wm} \tag{41}$$

$$C_R\sigma = \neg L_{G2}\sigma \vee \ldots \vee \neg L_{Gn}\sigma \vee \neg L_{W2}\sigma \vee \ldots \vee \neg L_{Wm}\sigma, \tag{42}$$

where L_{ij} represents a positive or negative literal. Note that in representing C_R as above, we have assumed (without loss of generality) that

$$\neg L_{G1}\sigma = L_{W1}\sigma.$$

Since N is a ramification of C_G,

$$\mathbf{W} \models (L_{G1} \wedge \ldots L_{Gn}) \supset \mathsf{N}. \tag{43}$$

So, the following instance of (43) holds:

$$\mathbf{W} \models (L_{G1}\sigma \wedge \ldots L_{Gn}\sigma) \supset \mathsf{N}\sigma]. \tag{44}$$

Rewriting (41) gives

$$L_{W2} \wedge \ldots \wedge L_{Wm} \supset \neg L_{W1}, \tag{45}$$

and therefore

$$\mathbf{W} \models (L_{W2}\sigma \wedge \ldots \wedge L_{Wm}\sigma \wedge L_{G2}\sigma \wedge \ldots \wedge L_{Gn}\sigma) \supset \neg L_{W1}\sigma. \tag{46}$$

But, $L_{G1}\sigma$ and $L_{W1}\sigma$ are identical literals, so

$$\mathbf{W} \models (L_{W2}\sigma \wedge \ldots \wedge L_{Wm}\sigma \wedge L_{G2}\sigma \wedge \ldots \wedge L_{Gn}\sigma) \supset L_{G1}\sigma. \tag{47}$$

Combining (47) and (44) gives the desired result:

$$\mathbf{W} \models (L_{W2}\sigma \wedge \ldots \wedge L_{Wm}\sigma \wedge L_{G2}\sigma \wedge \ldots \wedge L_{Gn}\sigma) \supset \mathsf{N}\sigma, \tag{48}$$

or equivalently,

$$\mathbf{W} \models \neg C_R \supset \mathsf{N}\sigma. \tag{49}$$

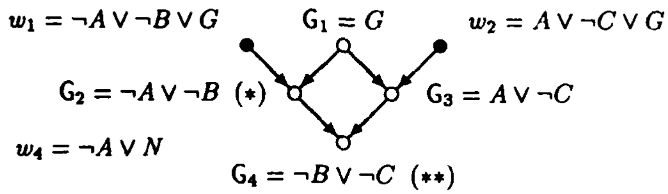
∎

4.7.2 Inheritance under G Factoring Steps

Theorem 4.9 *Suppose C_G is a clause from $\mathbf{G}^*$, and C_R is a factor of C_G via unifier σ. If N is a ramification of $\neg C_G$ then $\mathsf{N}\sigma$ is a ramification of $\neg C_R$.*

Proof: Let us represent the various clauses as follows:

$$C_G = \neg L_{G1} \vee \ldots \vee \neg L_{Gn} \tag{50}$$



(*) - N is ramification of G_2.
(**) - N is not a ramification of G_4.

Figure 10: Non-Inheritance of Ramifications

$$C_R\sigma = \neg L_{G2}\sigma \vee \ldots \vee \neg L_{Gn}\sigma, \tag{51}$$

where L_{ij} represents a positive or negative literal. Note that in representing C_R as above, it has been assumed (without loss of generality) that the factoring unifies only the first two literals, that is,

$$L_{G1}\sigma = L_{G2}\sigma. \tag{52}$$

Since N is a ramification of C_G

$$\mathbf{W} \models (L_{G1} \wedge \ldots L_{Gn}) \supset \mathsf{N}. \tag{53}$$

In addition the instance of (53) obtained by making substitution σ holds, so:

$$\mathbf{W} \models (L_{G1}\sigma \wedge \ldots L_{Gn}\sigma) \supset \mathsf{N}\sigma. \tag{54}$$

But, only one of the identical literals appears in (52), that is,

$$\mathbf{W} \models (L_{G2}\sigma \wedge \ldots L_{Gn}\sigma) \supset \mathsf{N}\sigma]. \tag{55}$$

So, from (55) the desired result follows, that is,

$$\mathbf{W} \models \neg C_R \supset \mathsf{N}\sigma. \tag{56}$$

∎

4.7.3 Inheritance under GG Resolution Steps

Ramifications are not necessarily inherited under GG-Resolution. Consider the following example:

Example 4.8 Figure 10 illustrates a goal that does not inherit the ramifications of its parents. Conisder a goal G and rules $A \wedge B \supset G$, $\neg A \wedge C \supset G$, and $A \supset N$. In clausal and non-clausal form, these are as follows:

	Clausal Form	**Non-Clausal**
G_1	$\neg G$	G
w_1	$\neg A \vee \neg B \vee G$	$A \wedge B \supset G$
w_2	$A \vee \neg C \vee G$	$\neg A \wedge C \supset G$
w_3	$\neg A \vee N$	$A \supset N$

WG-Resolution (Backchaining) gives two new goals

G_2	$\neg A \vee \neg B$	$A \wedge B$
G_3	$A \vee \neg C$	$\neg A \wedge C$

Performing GG-Resolution on G_2 and G_3 gives a new goal

G_4	$\neg B \vee \neg C$	$B \wedge C$

Note that G_2 has N as a ramification (via w_3). Goal G_2 gives rise to G_4 via GG-Resolution, but G_4 does *not* inherit N as a ramification (although some other line of reasoning may establish N as a ramification of G_4).

Theorem 4.10 *Suppose C_A and C_B are clauses from $\mathbf{G}^*$, C_R is a resolvent of C_A and C_B via unifier σ, and N is a ramification of $\neg C_A$. Suppose further that the literal on which C_A and C_B are resolved is not a member of all foundations of N, that is,*

$$\mathbf{W} \models (L_{A1} \wedge \ldots \wedge L_{A(i-1)} \wedge L_{A(i+1)} \wedge \ldots L_{An}) \supset \mathsf{N} \tag{57}$$

for $C_A = L_{A1} \wedge \ldots \wedge L_{An}$. Then $\mathsf{N}\sigma$ is a ramification of $\neg C_R$.

Proof: Let us represent the various clauses as follows:

$$C_A = \neg L_{A1} \vee \ldots \vee \neg L_{An} \tag{58}$$

$$C_B = \neg L_{B1} \vee \ldots \vee \neg L_{Bm} \tag{59}$$

$$C_R\sigma = \neg L_{A2}\sigma \vee \ldots \vee \neg L_{An}\sigma \vee \neg L_{B2}\sigma \vee \ldots \vee \neg L_{Bm}\sigma, \tag{60}$$

where L_{ij} represents a positive or negative literal. Note that in representing C_R as above, we have assumed (without loss of generality) that

$$\neg L_{A1}\sigma = L_{B1}\sigma.$$

The result follows immediately from (57). We know that the instance of (57) obtained by substitution σ must hold, that is,

$$\mathbf{W} \models (L_{A2}\sigma \wedge \ldots\ldots L_{An})\sigma \supset \mathsf{N}\sigma. \tag{61}$$

Note that (61) is derivable from (57) assuming that $i = 1$, that is, the first literal is the one resolved away. Since $\neg C_R$ contains a superset of the literals needed in (61) to prove $\mathsf{N}\sigma$, it must also be sufficient to prove $\mathsf{N}\sigma$. Thus we have,

$$\mathbf{W} \models (L_{A2}\sigma \wedge \ldots \wedge L_{An}\sigma \wedge L_{B2}\sigma \wedge \ldots \wedge L_{Bm}\sigma) \supset \mathsf{N}\sigma, \tag{62}$$

or

$$\mathbf{W} \models \neg C_R\sigma \supset \mathsf{N}\sigma. \tag{63}$$

∎

4.8 Related Work

4.8.1 McSkimin and Minker

McSkimmin and Minker [69] was an early effort at using semantic information to improve efficiency of database queries. Via information stored in a semantic network[14], McSkimmin and Minker (1) allowed unification of variables only with variables from the same domain (*semantic unification*), (2) checked to see if a query is inconsistent with a series of allowed forms (*semantic well-formedness*), and (3) based on information on the number of possible answers to a query, checked to see whether all the answers to a query had been found and therefore no more search is needed (*semantic actions*).

4.8.2 Stallman and Sussman

Stallman and Sussman's EL [90] was among the first to explicitly use forward reasoning in a declarative form to restrict a search.[15] EL's goals consisted of a set of variables to be bound in such a way as to be consistent with a model of various electrical circuit components. The values of the variables were currents, voltages, and states of transistors. The system had no backwards reasoning component or database lookup component, but rather had procedures

[14] The semantic network used by McSkimmin and Minker was very close to prediacte logic, however.

[15] Earlier, David Waltz [102] line-labeling program had propagated constraints via special-purpose procedures.

for guessing values of variables. After each guess, it reasoned forward to find ramifications of the existing bindings of its variables. Rather than allow random forward inference, EL's inference was limited to ***one-step deductions,*** that is, instantiation of Horn clauses of the form

$$A_1 \wedge \ldots \wedge A_m \supset N,$$

where all the A_i's were known to be true. After a new design decision was made, EL would perform all possible one-step deductions in an attempt to either show the design decisions to date to be inconsistent, or else to derive constraints on remaining circuit parameters.

4.8.3 MYCIN

While primarily a backwards inference system, in the MYCIN System (Shortliffe, Buchanan, et al [96,9,17]) it was beneficial to use a form of interleaved forward and backwards inference in certain cases. The backwards search of MYCIN possessed state information in what was called *contexts.* This state information was used in conjunction with forward reasoning for two purposes. First, the *preview mechanism* acted as a filter on rules, that is, if the premise of a rule could be immediately proven to be false, the rule could be eliminated from consideration. Consider a rule $A \wedge B \supset C$. Since large amounts of inference and (more importantly) interaction with the user might be involved in trying to establish A, it was critical to prune this rule if B was already known to be false. The second use of forward reasoning was similar to using ramifications. In order to smooth the interaction with users, it was necessary to ask questions in a fairly constrained fashion. One method for doing so was to force a set of questions to be asked whenever a context was instantiated. Antecedent rules were used to see whether answers to questions had already uniquely specified the answer to other questions, and thus, obviate the need to ask the certain questions. The forward reasoning was in the form of one-step deductions, as in EL, and results of one-step deductions could cause other one-step deductions to be triggered.

4.8.4 Stefik's MOLGEN

Mark Stefik's MOLGEN planner [93,92] worked in the domain of genetics experiment design. His basic notion was that in this domain, a hierarchical approach to experiment design would require very little backtracking if only the constraints created at any point are immediately propagated to the rest of the plan. At every point it was preferred to either make decisions for which only a single choice is possible or propagate a constraint rather than making guesses that might have to be retracted. Stefik dubbed such search control the *least commitment cycle.*

Constraint satisfaction and propagation in MOLGEN was done by LISP procedures for each constraint or constraint type. Each operator introduced into the plan could also introduce a set of constraints, and had the ability to regress or progress constraints through its action. One can view Stefik's constraints as a special class of prerequisite, a prerequisite that one should satisfy via as a side effect of other actions or variable choices in the plan rather than try to satisfy via introduction of special actions for this purpose.

4.8.5 King's QUIST

Jonathan King 's Ph.D. research [44] was embodied in a system called QUIST for "query optimization by semantic reasoning." The notion was that besides standard syntactic transformations on database queries, *semantic* restrictions upon the database could be used to reformulate a query as a less expensive query. For example, if all ships above a certain tonnage is known to be supertankers, and it is cheaper to find all supertankers than all ships, this fact could be used in answering queries about ships. King defined the notion of *semantic equivalence transformations*, transformations of a query Q to a query Q' such that are not logically equivalent, but for every *permitted* interpretation, the two queries have the same set of answers.

In order to generate semantically equivalent queries, QUIST defined a set of transformations allowing the system to do one-step deductions and manipulate conjunctions and disjunctions. The process produced new, semantically equivalent queries that could be cheaper to solve by the same mechanisms outlined in Section 3.6. Estimates of costs of queries were established by working with well-defined models of the database and queries. It considered only the class of restrict-join-project queries, an incomplete but very useful class of relational database queries, and used a simple model of access and storage (based in the work of Blasgen and Eswaren [3] at IBM) to measure the cost of processing a given query.

4.8.6 Kohli and Minker

In [45], Madhur Kohli and Jack Minker proposed controlling backwards search by using integrity constraints.[16] Their paper dealt with logic programs written in function-free ordered Horn clauses, and it assumed the presence of integrity constraints on the database, also expressed as Horn clauses. Like Ordered Resolution on HOH-clauses (See Section 2.4), and in contrast to Prolog, the search strategy is not restricted to be depth-first. The paper

[16] In Kohli and Minker's terminology, the roles of "forward" and "backward" are reversed.

proposed checking every goal generated against all integrity constraints, the pruning of inconsistent goals, and notions similar to generators and filters.[17] Kohli and Minker cached ramifications they derived ("implicit integrity constraints" in their terminology), and because only Horn clauses were considered, all ramifications are inherited from parent goals (See Section 4.7).

4.8.7 Chakravarathy, et al

In [12,14,13] U. S. Chakravarthy, Jack Minker and their colleagues have described the extension of Semantic Query Optimization to deal with deductive databases.[18] In Chakravarthy's system, it is assumed that there will be many, many queries and a fairly limited number of integrity constraints. Rather than trying to optimize each query as King does, Chakravarthy's system stores appropriate supersumptions with each intensional and extensional relation. When a query is presented to the system, it may then quickly see which of its cached supersumptions are appropriate. Given the huge potential speedup on large databases, it is worth the overhead of deriving and storing supersumptions with each relation. Note that it is usually combinations of relations in a query that make supersumptions useful. To cache all potentially useful supersumptions for each relation is combinatorically explosive, but given Chakravarthy's assumptions about the number of queries versus the number of relations and integrity constraints, it is reasonable to do so.

Chakravarthy starts with integrity constraints and intensional and extensional database definitions stated in clausal form. To find potentially useful supersumptions, Chakravarthy looks at *partial subsumptions* of definitions of relations, that is, definitions that are subsumed by a subset of literals in an integrity constraint. Using subsumption rather than simple unification forces the supersumption to hold for all values of all variables. The treatment is quite similar to the $\mathcal{P}_{RPS}$ Procedure of this thesis (See Section 4.6, and the name $\mathcal{P}_{RPS}$ "resolution with partial subsumption" was chosen to point out the similarity with Chakravarthy's approach to database optimization.

A number of other researchers have consider semantic query optimization on databases, including Hammer and Zdonik [35], Xu [107] and Jarke et al [43].

---

[17] It is not clear to the present author that the mechanism proposed for generators is guaranteed to produce only correct answers.

[18] As described in Reiter [75], databases are divided into *extensional* and *intensional* relations, where extensional relations are relations that can be looked up in the database, whereas intensional relations must be reduced to combinations of intensional relations. The reduction of intensional relations takes place via rules from which the extensional definition of an intensional relation may be deduced — hence the name *deductive database*.

4.8.8 Lee, et al.

Theorem 4.2, Section 4.5 shows that although resolution in not deductively complete, it can deduce a clause that subsumes any clause implied by the base set. Although the proof was done independently, it turns out that a virtually identical result was published exactly 20 years ago in Richard C. T. Lee's PhD thesis [50] and extended to linear resolution (of which set of support is a special case in this case) by Minicozzi and Reiter [70] in 1972. Thus, it is only fair to view Theorem 4.2 and Theorem 4.6.3 as very minor variations on these earlier results. Thanks to Richard Waldinger and Mark Stickel for pointing out this research to me.

4.9 Summary

This chapter has explored deduction of ramifications from a conjunctive goal $G = G_1 \wedge \ldots \wedge G_m$, where the G_i are atomic formulas. Two resolution-based procedures, $\mathcal{P}_{RGC}$ and $\mathcal{P}_{RPS}$ are shown to be complete for this problem, though not in the usual sense of generating all possible ramifications. Instead they are complete in that for any ramification N, both of these procedures will generate a ramification that is at least as "strong" as N. More precisely, for every clause N_i in a conjunct normal-form of N, $\mathcal{P}_{RGC}$ and $\mathcal{P}_{RPS}$ can generate a clause N_i' that subsumes N_i. $\mathcal{P}_{RPS}$ is superior to $\mathcal{P}_{RGC}$ in that it is completely within the framework of resolution on the world model $\mathbf{W}$, that is, $\mathcal{P}_{RPS}$ is a restriction on resolutions on base set $\mathbf{W}$ that disallows resolutions unrelated to the goal G at hand, but still allows all needed ramifications to be found.

In database retrievals, a single goal G is given and variable bindings for x must be found such that G holds. As discussed in Chapter 2, the problems considered here involve goal reduction, that is, reduction of the orignal goal to other goals via backwards reasoning. Because ramifications could be deduced from any such goal, it is important to consider whether the search for ramifications of one goal can be used in the search for ramifications of another goal. There are two forms of sharing of search explored in this chapter: (1) *Inheritance* of ramifications from parent goals, and (2) *Caching* of ramifications along with the goal conjuncts from which they were derived. Section 4.7 derived results for what ramifications are and are not inherited from parents. A simple modification of $\mathcal{P}_{RGC}$ for caching ramifications and the conjuncts on which they are based is presented in Section 4.5.3. $\mathcal{P}_{RPS}$, on the other hand, handles the caching of ramifications in a much more natural fashion, at all times recording the goal conjuncts (and only those goal conjuncts) on which a deduction of a ramification is based.

Chapter 5

Conclusion

Synthesis problems constitute a major class of problems encountered in many fields. Robot planning, circuit design, automatic generation of diagnostic tests, program synthesis, and automatic theorem proving are among the synthesis problems commonly encountered in the AI literature. Automatic design synthesis has been of interest since the very beginnings of AI. *Deductive* approaches to synthesis problems, that is, constructing a design as part of *the proof of a theorem*, date from the work of Green [32] and Waldinger and Lee [99] in the late 1960's.

The current research continues in the tradition of deductive design synthesis. In previous deductive synthesis, design has been a process of backwards reasoning from a goal formula, representing the design as a term in the logic. In reasoning only backwards during the design process, such systems have not considered interactions of various parts of the design already specified with each other and with the remaining subgoals. In addition, representation of the design as a term, that is, a composition of functions, has made it unnatural to reason *about* the design, and has limited the set of design decisions that can easily be expressed.

This thesis has two main themes. First, for reasonable behavior over a wide spectrum of goals, *the design process should be bidirectional*; one should reason backwards from the goal (goal reduction) and forward from the goal and any design decisions that have been made (consistency checking and/or supersumption). Second, *designs should be represented as formulas* rather than as terms. By doing so one gains expressiveness in representing design decisions and the ability to reason directly about the design.

The main points of this thesis are summarized in the following sections. The first presents the main contributions of this thesis. The second section summarizes its main limitations, and the third suggests directions for future work in this area and improvements of this thesis.

5.1 Summary of Contributions

5.1.1 A Framework for Design

Chapter 2 defined a *residue*, a new definition for specification of a design object. In this formulation, both complete and incomplete designs are represented as single formula of first-order predicate calculus. A formula is a residue, i.e., a legal design, if it (1) logically implies the goal specification, (2) is consistent with the set of axioms describing the world, and (3) is a conjunction of formulas, each of which can be assumed to be achievable in the world being modeled.

The chief advantage of Residue's approach is that the system can use the entire set of relations of the logic to be used to express constraints on the design. The ability to express all desired design constraints is crucial during the design process, when imposition of an unnecessarily strong constraint can lead to needless backtracking. In contrast, representation of the design as a term limits the expressible design constraints to those for which there is a single pre-existing function.

Consistency checking corresponds to seeing that all the constraints imposed upon the design can be realized at once, an integral part of complicated design problems. Without the ability to check consistency of a design, one may not partially specify a component on which other, possibly inconsistent, constraints will later be imposed. For a design represented as a formula, this notion corresponds to consistency of the design formula with the set of axioms describing the world. In contrast, checking consistency of a design expressed as a single term is an ad hoc process. There is no general way to use a set of axioms describing the world to check whether the object denoted by a given term is consistent with that set of axioms. As a result, single-term approaches have usually been limited to design problems in which consistency checking is not needed, a severe limitation.

5.1.2 Procedure for Design Synthesis

Chapter 2 also described two procedures, Resolution Residue and Ordered Residue, for generation of residues. Both of these procedures work by reducing a goal specification via backwards reasoning to a set of primitively achievable specifications. Resolution Residue uses binary resolution for its backwards inference, and Ordered Residue uses an ordered resolution on Horn clauses.

For both residue procedures, appropriate completeness results were proven. The completeness results show that although not every residue can be generated, for every residue not generated, a residue at least as general will be generated.

5.1.3 Supersumption

This research has dealt with the derivation and use of constraints derived from a goal (or subgoal) via forward reasoning. Such constraints are called *ramifications* and the process of using those constraints is called *supersumption*. Supersumption captures the notion that one should not only consider the consistency of a partially specified solution, but should be able to make use of any conditions *necessary* for consistency.

The Residue Procedure allows pruning of inconsistent designs during the design procedure. Supersumption is a way to (1) avoid generating some of those inconsistent designs, and (2) avoid some of the overhead of consistency checking over a large set of designs.

5.1.4 Procedure for Finding Ramifications

Chapter 4 presented two procedures for finding *ramifications*, constraints necessary for consistency of a set of design constraints and any remaining subgoal. Both of these procedures find ramifications as a part of of the checking consistency process.

The first procedure $\mathcal{P}_{RGC}$ uses resolution as a deduction procedure rather than its usual use as a refutation procedure. Although resolution is not deductively complete, the completeness results of this chapter show that for every ramification not derivable by resolution, a ramification that is at least as strong is derivable. In addition, $\mathcal{P}_{RGC}$ uses conjuncts of the goal as a set of support from which to perform resolutions. This has the important property that any ramification that is derivable is derivable without resolving random facts about the world with each other — all ramifications are derivable from goal conjuncts.

The second of the two procedures, $\mathcal{P}_{RPS}$, has all the above properties of the $\mathcal{P}_{RGC}$ with one major difference: $\mathcal{P}_{RPS}$ derives ramifications in the form of facts that may be directly added to the database of axioms about the world. Such facts record the ramification derived, and the goal conjuncts that were needed to derive that ramification. By recording such facts, other goals with conjuncts in common (modulo variable names) may use the result of the previous derivation of ramifications.

5.2 Main Limitations of the Approach

5.2.1 Assumable Formulas must be Atomic

Chapter 2 makes the assumption that the design will consist of a set of primitively achievable *atomic* formulas. Disjunctions and conditions are thus excluded from designs. This assumption appears both in the completeness theorems of Chapter 2 as well as in the Resolution Residue and Ordered Residue procedures. In these procedures, designs are represented as

the negation of a single clause, that is, a design is a conjunction of literals. To represent a design containing disjunctions, multiple clauses would have to be used for the design.

The severity of this restriction is not entirely clear. At first glance it would appear that conditional plans would be impossible to build as residues, but in fact they have been synthesized. To do so, it was necessary only to represent actions as mappings to one of several possible succeeding states based upon the outcome of that action. It is as yet unknown whether procedures can be found for generating non-atomic residues, and under what circumstances it is desirable to generate designs out of such components.

5.2.2 Design and Subdesigns Have No Name

The Residue Procedure, in contrast to the single-term approach, does not reify the design; there is no perfectly natural way to refer to the design as a whole, nor is there a way to refer to portions of the design. Instead, the set of design decisions comprising the design are stated as facts describing the entire world. For example, it is awkward to say that the cost *of the design* must be less than \$1.00. One can probably avoid this problem by introducing a relation on design components and designs (or subdesigns) such that the relation holds for every component of a particular design (or subdesign), but to date, use of such axiomatizations has not been explored.

5.2.3 Rederivation of Cached Deductions

A good deal of the power of supersumption comes from its caching of ramifications; the cached ramifications can be used to filter (prune) other design candidates without the ramification being rederived. In this thesis, as in many other systems, the caching of the result of a series of deductions does not mean that those deductions will not be performed again as forward reasoning from another goal.

In systems for which it is important to preserve completeness, it is a difficult problem knowing when old results can be reused without further exploration of the path on which they were found. The old path might not have been completely explored, or new facts might enable derivation of new results that were not possible when the path was previously explored.

5.3 Further Work

5.3.1 Control Heuristics for Residue

Besides the usual search control issues in searching a search space, Residue involves both forward reasoning (consistency checking) and backwards reasoning (goal reduction). Although it is not logically necessary to check consistency of a design until it is complete, one of the major motivations for Residue is the ability to quickly prune inconsistent designs. To date, there has not been work on when consistency checking is heuristically appropriate during the design process. Similarly, there is little known on how much effort to spend checking consistency at any given point. Such a decision will depend upon such factors as the branchiness of an upcoming decision point, how close to being overconstrained the design currently is, and the extent to which the upcoming decision is perceived to further constrain the set of design candidates.

5.3.2 Cost of Solving a Problem

In order to know when to use a given supersumption, one must be able to estimate the cost of finding a solution to a given goal. To date, we have good models only under strong sets of assumptions. In particular, we are good at estimating the cost of lookups on conjunctive queries in extensional databases. For subgoals whose solutions involve backwards reasoning, there is still very little work.

5.3.3 Control Heuristics for Finding Ramifications

As was discussed in Chapter 4, ramifications can be found as part of the process of consistency checking. If one checks consistency of partially completed designs, there is a tradeoff of forward and backwards reasoning (as mentioned above in Section 5.3.1). A similar tradeoff exists when forward reasoning is used to find ramifications for supersumption; good heuristics to decide when to do such forward reasoning and what forward reasoning are necessary. There may be cases where it would pay off to look for ramifications, whereas consistency checking would not be called for.

5.3.4 Probable Constraints

The above research has involved finding *ramifications* of a goal. As was mentioned in Chapter 3, supersumption can be used with other constraints as well. For problems in which not all solutions to a problem are needed, use of probable constraints, that is, constraints

derived by *plausible* reasoning from a goal, is at least as useful as supersumption using only ramifications.

The general use of probable constraints is equivalent to heuristic search. Finding probable constraints by plausible reasoning from goals (and subgoals) and the addition of such constraints to the goal at hand is a very specific form of heuristic search guidance. It has the same control problems as supersumption with ramifications, and heuristics for search control applicable to supersumption with ramifications should be applicable to supersumption with probable constraints as well.

Bibliography

[1] Avron Barr, Paul R. Cohen, and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volumes I, II, and III.* William Kaufmann, Inc., Los Altos, CA, 1981 and 1982.

[2] D.R. Barstow. An experiment in knowledge-based automatic programming. *AI*, 12(2):73–119, August 1979.

[3] M. W. Blasgen and K. P. Eswaren. Storage access in relational databases. *IBM Systems Journal*, 16(4), 1977.

[4] W. W. Bledsoe. Non-resolution theorem proving. *AI*, 9(1):1–35, August 1977.

[5] R.S. Boyer. *Locking: A Restriction of Resolution.* PhD thesis, University of Texas at Austin, August 1971.

[6] R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[7] R.S. Boyer and J S. Moore. Proving theorems about Lisp programs. In *Advance Papers from the Third IJCAI*, SRI, Menlo Park, CA, Stanford Univ., Stanford, CA, August 1973. Revised version appears in JACM, Volume 22, Number 1, pages 129-144, 1975.

[8] B.G. Buchanan and E. A. Feigenbaum. Dendral and meta-dendral: their applications dimension. *Artificial Intelligence*, 11:5–24, August 1978. Special issue on Applications to the Sciences and Medicine. Edited by N.S. Sridharan.

[9] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA, 1984.

[10] Bruce Buchanan, Russ Altman, James Brinkley, Craig Cornelius, Bruce Duncan, Barbara Hayes Roth, Michael Hewett, Olivier Lichtarge, and Oleg Jardetzky. *The Heuristic Refinement Method for Deriving Solution Structures of Proteins.* KSL-85-41, STAN-CS-86-1115, Knowledge Systems Laboratory, Stanford University, March 1986. Also appears in *Proceedings of the National Academy of Science.*

[11] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

[12] U. S. Chakravarthy. *Semantic Usage and Query Optimization in Deductive Databases.* Technical Report TR-1413, University of Maryland, June 1984.

[13] U. S. Chakravarthy, J. Minker, and J. Grant. Semantic query optimization: additional constraints and control strategies. In *Proceedings of the First International Conference on Expert Database Systems*, pages 259–269, Charleston, South Carolina., April 1986.

[14] U. S. Chakravarthy. *Semantic Query Optimization in Deductive Databases.* PhD thesis, University of Maryland, July 1985.

[15] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[16] Eugene Charniak and Drew V. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts., 1985.

[17] W.J. Clancey. Classification problem solving. *Proceedings of the National Conference on Artificial Intelligence*, 49–55, 1984.

[18] W. S. Cooper. Fact retrieval and deductive question answering information retrieval systems. *Journal of ACM*, 11:117–137, April 1964.

[19] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[20] Johan de Kleer. *An Assumption-Based TMS.* Technical Report, XEROX Palo Alto Research Center, February 1985.

[21] Johan de Kleer. Choices without backtracking. In *Proceedings of the National Conference on Artificial Intelligence*, AAAI, University of Texas at Austin, TX, August 1984.

[22] John Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[23] George W. Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, New York, 1969.

[24] R.E. Fikes. Ref-arf: a system for solving problems stated as procedures. *AI*, 1(1):27–120, Spring 1970.

[25] R.E. Fikes and N.J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[26] R.E. Filman and R.W. Weyhrauch. *An FOL Primer: AI Lab.* Memo 288, Stanford University, 1976.

[27] M. R. Genesereth. *An Overview of MRS.* Heuristic Programming Project Memo, Stanford University, June 1983.

[28] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24, 1984.

[29] Randy Goebel, Koichi Furukawa, and David Poole. *Using definite clauses and integrity constraints as the basis for a theory formation approach to diagnostic reasoning.* Research Report CS-85-50, University of Waterloo, December 1985.

[30] Claude Cordell Green. *The Application of Theorem Proving to Question-Answering Systems*. PhD thesis, Stanford University, June 1969. (also published as Stanford AI Project Memo AI-96, and by Garland Publishing, New York, 1980).

[31] Cordell C. Green. Application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.

[32] Cordell C. Green. Theorem proving by resolution as a basis for question-answering systems. In Meltzer and Michie, editors, *Machine Intelligence 4*, Edinburgh University Press, Edinburgh, 1969.

[33] Cordell C. Green and S.J. Westfold. Knowledge-based programming and deduction in algorithm design. In *Machine Intelligence 10*, pages 339–359, Ellis Horwood Limited, Chichester, England, 1982.

[34] Russell Greiner. *Learning by Understanding Analogies.* PhD thesis, Stanford University, September 1985. Technical Report STAN-CS-1071.

[35] M. H. Hammer and S. B. Zdonik. Knowledge based query processing. In *Proceedings of the Sixth International Conference on Very Large Databases*, pages 137–147, September 1980.

[36] Charles Hartschorne and Paul Weiss, editors. *Collected Papers of Charles Sanders Peirce.* MIT Press, Cambridge, Mass., 1933.

[37] Patrick J. Hayes. In defence of logic. In *Proceedings of the Fifth IJCAI*, pages 559–565, Cambridge, MA, August 1977.

[38] P.J. Hayes. A logic of actions. In *Machine Intelligence 6*, pages 495–520, American Elsevier, New York, 1971.

[39] P.J. Hayes. Robotologic. In *Machine Intelligence 5*, pages 533–554. Edinburgh University Press, Edinburgh, Scotland, 1969.

[40] Barbara Hayes-Roth, Bruce Buchanan, Olivier Lichtarge, Michael Hewett, Russ Altman, James Brinkley, Craig Cornelius, Bruce Duncan, and Oleg Jardetzky. Deriving protein structure from constraints. In *Proceedings of the National Conference on Artificial Intelligence*, pages 904–909, Philadelphia, Pennsylvania, 1986. Also appears as Stanford Technical Report KSL-86-51.

[41] Barbara Hayes-Roth, Frederick Hayes-Roth, Stanley J. Rosenschein, and Stephanie Cammarata. Modeling planning as an incremental, opportunistic process. In *Proceedings of the Sixth IJCAI*, pages 375–383, 1979.

[42] C. G. Hempel. *Aspects of Scientific Explanation.* Free Press, New York, 1965.

[43] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing prolog front-end to a relational query system. In *Proceedings of ACM SIGMOD*, pages 296–306, June 1984.

[44] Jonathan J. King. *Query Optimization by Semantic Reasoning.* PhD thesis, Department of Computer Science, Stanford University, May 1981.

[45] Madhur Kohli and Jack Minker. Intelligent control using integrity constraints. In *Proc. of the Nat'l Conf. on AI*, AAAI, Washington, D.C., August 1983.

[46] K. Konolige. *A deduction model of belief and its logics.* PhD thesis, Stanford University, June 1984. also report no. STAN-CS-84-1022.

[47] R. Kowalski. Algorithm = logic + control. *CACM*, 22(7):424–436, 1979.

[48] R. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.

[49] R. C. T. Lee, C. L. Chang, and R. J. Waldinger. An improved program-synthesizing algorithm and its correctness. *CACM*, 17(4):211–217, April 1975.

[50] Richard C. T. Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms.* PhD thesis, University of California at Berkeley, 1967.

[51] D. W. Loveland. A linear format for resolution. In *Proc. IRIA Symp. Automatic Demonstration*, pages 147–162, Springer-Verlag, Versailles, France, 1968.

[52] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland Publishing Co., Amsterdam, 1978.

[53] D. Luckham. Refinements in resolution theory. In *Proc. IRIA Symp. Automatic Demonstration*, pages 163–190, Springer-Verlag, Versailles, France, 1968.

[54] David Luckham and Nils J. Nilsson. Extracting information from resolution proof trees. *Artificial Intelligence*, 2:27–54, 1971.

[55] Jock Mackinlay and Michael R. Genesereth. Expressiveness of languages. In *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas, August 1984.

[56] Jock D. Mackinlay. *Automatic Design of Graphical Presentations.* PhD thesis, Stanford University, December 1986.

[57] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill Inc., San Francisco, 1974.

[58] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[59] Zohar Manna and Richard Waldinger. Special relations in automated deduction. *Journal of ACM*, 33(1):1–59, 1986.

[60] Zohar Manna and Richard Waldinger. Synthesis: dreams $\Rightarrow$ programs. *IEEE Transactions on Sortware Engineering SE-5*, 4:294–328, 1979. Also published as Stanford A.I. Lab Memo 302, 1977.

[61] J. P. Martins. ***Reasoning in Multiple Belief Spaces.*** Technical Report Computer Science Department Technical Report 203, MIT, Buffalo, N.Y., 1983.

[62] D. McAllester. ***An Outlook on Truth Maintenance.*** Technical Report MIT AIM-551, MIT, Cambridge, Mass., 1980.

[63] D. McAllester. ***Reason Utility Package User's Manual.*** Technical Report MIT AIM-667, Massachusets Institute of Technology, Cambridge, Mass., 1982.

[64] D. McAllester. ***A Three-Valued Truth Maintenance System.*** Technical Report S.B. Thesis, Department of Electrical Engineering, Massachusets Institute of Technology, Cambridge, Mass., 1978.

[65] John McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the Fifth IJCAI*, pages 1038–1044, Cambridge, 1977.

[66] John McCarthy. *Situations, Actions, and Causal Laws.* Stanford AI Lab Memo AIM-2, Stanford University, 1963. Also published in *Semantic Information Processing* (Marvin Minsky, ed.), MIT Press, Cambridge, Mass., 1968, pages 410-417.

[67] Drew McDermott. Contexts and data dependencies: a synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 1983.

[68] Drew McDermott. A critique of pure reason. Yale University, June, 1986, unpublished.

[69] J. R. McSkimin and Jack Minker. The use of a semantic network in a deductive query answering system. In *Proceedings of the Fifth IJCAI*, pages 50–58, Cambridge, Massachusetts, August 1977.

[70] Eliana Minicozzi and Raymond Reiter. A note on linear resolution strategies in consequence finding. *Artificial Intelligence*, 3:175–180, 1972.

[71] Allen Newell, John C. Shaw, and Herbert A. Simon. *Preliminary Description of General Problem Solving Program-I (GPS-I).* Report CIP Working Paper 7, Carnegie Instit. of Tech., Pittsburgh, PA, 1957.

[72] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, 1980.

[73] David Poole, Randy Goebel, and Romas Aleliunas. *Theorist: a logical reasoning system for defaults and diagnosis.* Research Report CS-86-06, University of Waterloo, February 1986.

[74] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[75] Raymond Reiter. On closed world data bases. In H. Gaillaire and Jack Minker, editors, *Logic and Data Bases*, Plenum Press, New York, 1978.

[76] J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[77] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[78] P. Roussel. *Prolog: Manual de reference et d'utilisation.* 1975. Groupe d'Intelligence Artificielle, Marseille-Luminy; September.

[79] Stuart Russell. *The Compleat Guide to MRS.* KSL-85-12, Stanford University Computer Science Department, June 1985.

[80] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[81] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Technical Report Technical Note 109, SRI, August 1975.

[82] E.H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* Elsevier North Holland, Inc., New York, 1976. based on Phd thesis, Stanford University, Stanford, CA, 1974.

[83] Narinder Singh. *Exploiting Design Morphology to Manage Complexity.* PhD thesis, Stanford University, August 1985. Published by Schlumberger Palo Alto Research.

[84] J.R. Slagle. Experiments with a deductive question answering program. *CACM*, 8, December 1965.

[85] D. E. Smith. *Controlling Inference.* PhD thesis, Stanford University, August 1985.

[86] D. E. Smith and M. R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 25, 1985.

[87] Douglas R. Smith. Derived preconditions and their use in program synthesis. In *6th Conference on Automatic Deduction*, pages 172–193, New York, 1982.

[88] Douglas R. Smith. A problem reduction approach to program synthesis. In *Proceedings of the Eighth IJCAI*, pages 32–36, IJCAI, Karlsruhe, West Germany, August 1983. Vol. 1.

[89] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1), 1985.

[90] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, October 1977. Reprint in "AI-MIT", vol. 1, pp.31-91. Also MIT AI Memo 380,'76.

[91] Guy L. Steele. *The Definition and Implementation of a Computer Programming Language based on Constraints*. Technical Report Artificial Intelligence Laboratory, TR-595, MIT, Cambridge, Mass., 1979.

[92] Mark J. Stefik. Planning with constraints. *Artificial Intelligence*, 16(2):111–140, 1981.

[93] Mark Jeffrey Stefik. *Planning with Constraints*. PhD thesis, Computer Science Department, Stanford University, January 1980. Stanford Rep. Nos. HPP-80-2, STAN-CS-80-784.

[94] G. J. Sussman. *A Computational Model of Skill Acquisition*. Technical Report MIT AI-TR-297, NIL, Cambridge, Mass., August 1973.

[95] Richard Treitel and Michael R. Genesereth. Ordered resolution with negative replacement. June 1986. Personal Communication.

[96] W. van Melle. *A domain-independent system that aids in constructing knowledge-based consultation programs*. PhD thesis, Computer Science Department, Stanford University, June 1980.

[97] Richard Waldinger. Achieving several goals simultaneously. In Edward W. Elcock and Donald Michie, editors, *Machine Intelligence 8: Machine Representation of Knowledge*, pages 94–136, Ellis Horwood Ltd., Chichester, England, 1977. Also SRI Technical Note 107, July, 1975.

[98] R.J. Waldinger. *Constructing Programs Automaically Using Theorem Proving.* Ph D. Dissertation, Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA, 1969. AD 697 041.

[99] R.J. Waldinger and R.C.T. Lee. Prow: a step toward automatic program writing. In Walker, editor, *International Joint Conference on Artificial Intelligence*, pages 241–252, Mitre Co., 1969.

[100] Adrian Walker. Prolog/ex1, an inference engine which explains both yes and no answers. In *Proceedings of the Eighth IJCAI*, pages 526–528, IJCAI, Karlsruhe, West Germany, August 1983. Vol. 1.

[101] Adrian Walker. *Prolog/Ex1, An Inference Engine Which Explains Both Yes and No Answers.* Technical Report RJ3771, IBM Research Laboratory, San Jose, California, January 1983.

[102] D. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91, McGraw-Hill, New York, 1975. based on Phd thesis, MIT, Cambridge, MA, 1972.

[103] David L. Waltz. *Generating Semantic Descriptions from Drawings of Scenes with Shadows.* Technical Report MIT AI-TR-271, MIT, Cambridge, Mass., November 1972.

[104] T. Winograd. Frame representations and the procedural/declarative controversy. In *Representation and Understanding: Studies in Cognitive Science*, pages 185–210, Academic Press, New York, 1975.

[105] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12:536–541, 1965.

[106] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984.

[107] G. D. Xu. *Search Control in Semantic Query Optimization.* Technical Report 83-9, University of Massachusetts, Department of Computer Science, Amherst, 1983.